INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN OF LOW-DENSITY PARITY-CHECK CODES FOR MAGNETIC RECORDING CHANNELS

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Richard M. Todd

Norman, Oklahoma

2002

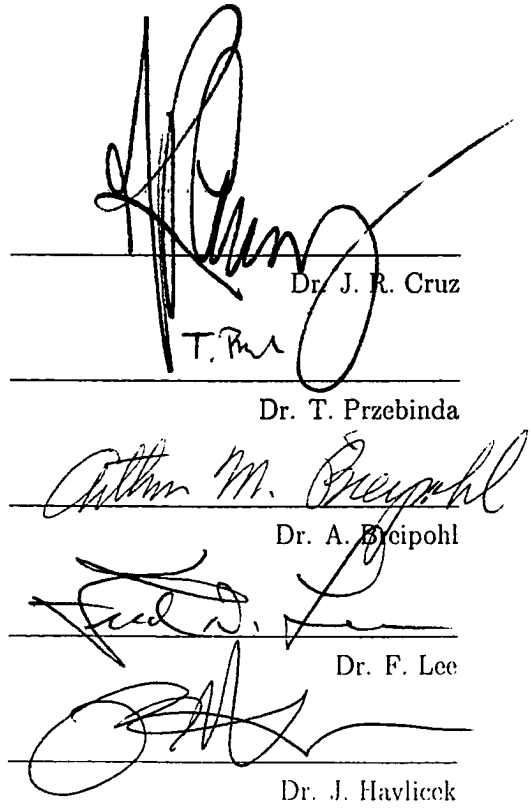
UMI Number: 3070645

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

DESIGN OF LOW-DENSITY PARITY CHECK CODES FOR MAGNETIC RECORDING CHANNELS

A Dissertation APPROVED FOR THE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



Acknowledgements

I would like to thank my parents for their support. I would also like to thank Robert Shull, Raymond Schlecht, and Mike Callahan for helping me learn Unix all those years ago. I would also like to thank Dr. Henry Neeman of the OU Supercomputing Center for Education and Research for helping me parallelize and optimize my code and allowing me use of the center's machines.

Contents

List of Tables

List of Figures

Abstract

DESIGN OF LOW-DENSITY PARITY-CHECK CODES FOR MAGNETIC RECORDING CHANNELS

Richard M. Todd

Advisor: Dr. J. R. Cruz

A technique for designing low-density parity-check (LDPC) error correcting codes for use with the partial-response channels commonly used in magnetic recording is presented. This technique combines the well-known density evolution method of Richardson and Urbanke for analyzing the performance of the LDPC decoder with a newly developed method for doing density evolution analysis of the Bahl-Cocke-Jelinek-Raviv (BCJR) channel decoder to predict the performance of LDPC codes in systems that employ both LDPC and BCJR decoders, and to search for good codes. We present examples of codes that perform 0.3dB to 0.5dB better than the regular column weight three codes employed in previous work.

A new algorithm is also presented, which we call "MTR enforcement". Typical magnetic recording systems employ not just an error correcting code, but also some form of run-length-limited code or maximum-transition-run (MTR) code. The MTR enforcement algorithm allows us to exploit the added redundancy imposed by the MTR code to increase performance over that of a magnetic recording system which does not employ the MTR enforcer. We show a gain of approximately 0.5dB from the MTR enforcer in a typical magnetic recording system. We also discuss methods of doing so-called "soft-error estimates", which attempt to extrapolate the bit-error-rate (BER) curve from Monte Carlo simulations down below the limits for which the traditional BER results are valid. The recent work by Yedidia on generalizations of the belief propagation algorithm is discussed, and we consider problems that arise in using this generalized belief propagation method for decoding LDPC codes.

Chapter 1

Introduction to Magnetic Recording of Data

The basic idea of magnetic recording of data, as done in most disk drives today, is to store data on a magnetizable medium in such a fashion that it can be practically recovered later. The core of any system for magnetic data recording is the magnetic recording head, such as can be seen in Fig. 1.1. When reading or writing data, the head flies over the magnetic medium at some velocity v. For writing data, a current is sent through the head in either the forward or reverse direction for a time period T , i.e., the current $i(t)$ equals either $+A$ or $-A$ for some current value A. Each time period T thus allows the storage of one bit of data. The current causes the magnetic medium passing under it to be fully magnetized in one direction or the other, hence storing the sequence of $+A$ and $-A$ current values as magnetized regions of length vT. This is called **saturation recording**, because the medium is saturated or fully magnetized, as opposed to, say, audio recording on analog cassette recorders, where the medium can be in intermediate levels of magnetization. We also call this **longitudinal recording**, because the regions of magnetization have their long axis along the direction of travel of the head. (There are other possible techniques of magnetic recording which work differently, such as perpendicular recording. However, perpendicular recording systems are not used in disk drives on the market today, though this may change in the future.) When reading the data, the head passes over the magnetized regions and this induces a current in the head from which our magnetic recording system can infer the original sequence of data that was written.

1.1. Inter-Symbol Interference and the Lorentzian Channel Model

If the current produced by the head when reading were a simple function of the original write sequence (e.g., just some multiple of the original write current sequence

of $+A$ and $-A$ values), magnetic recording system design would be very easy. Unfortunately, things are not so simple. For one thing, the read head responds only to *changes* in the magnetization, not to the actual magnetization itself, so that current pulses are produced only when the magnetization changes, i.e., when the original write current changes from $+A$ to $-A$ or vice versa. Also, the current pulse is not a simple delta function, but rather a function which can have significant non-zero values over time-spans much longer than our bit time T. This means that during any given interval $t \in [nT, (n+1)T)$, the read current includes contributions from not just the current bit under the read head, but from several adjacent bits as well. The contributions from the other bits interfere with those from the current bit; this is called **inter-symbol interference**.

To a fairly good approximation, in a longitudinal recording system, the resulting current pulse from a single magnetization change at location $x = 0$ on the magnetic medium is a multiple of the **Lorentzian response**, given by the following function of position x of the head over the medium:

$$L(x) = \frac{1}{1 + \left(\frac{2x}{\ell}\right)^2} \tag{1.1}$$

where ℓ is a length parameter determined by the magnetic characteristics of the medium. Given that our head is moving over the magnetic medium at some velocity v, and supposing that at time $t = 0$ our head is at position $x = 0$ we have the Lorentzian channel response as a function of time

$$L(t) = \frac{1}{1 + \left(\frac{2vt}{\ell}\right)^2} \quad . \tag{1.2}$$

Now, if we were to write further transitions on the medium at times $t = nT$, we would be creating magnetization regions of length vT upon the magnetic medium. We conventionally define a dimensionless parameter S, called the **channel density**,

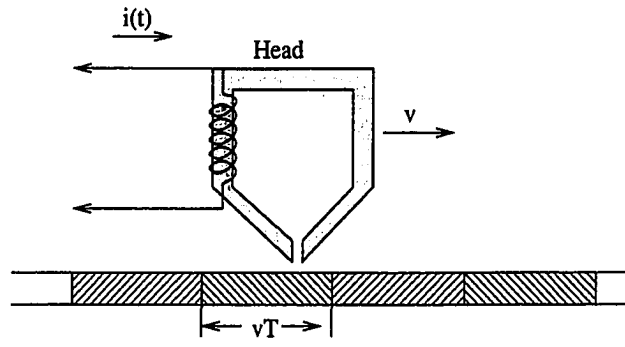


Figure 1.1: Drawing of magnetic recording head and medium.

as the ratio of the characteristic length ℓ and the length of our magnetization regions vT:

$$S = \frac{\ell}{vT} \quad . \tag{1.3}$$

The higher S is, the smaller vT is and the more tightly packed and smaller the magnetization regions are on the magnetic medium. We can now rewrite (1.2) in terms of the channel density and our bit time T as

$$L(t) = \frac{1}{1 + \left(\frac{2t}{ST}\right)^2} \quad . \tag{1.4}$$

ST is sometimes called the pulse-width-half-magnitude parameter, since $L(t) = 1/2$ at $t = \pm ST/2$, so the width of the region for which $L(t)$ has half or more of its peak magnitude is ST. Typical values of the channel density S for current magnetic recording systems are around $S = 3$. An example of what the Lorentzian response for $S = 3$ looks like is shown in Fig. 1.2 for $T = 1$, so one can see that the response from each bit affects the output in several adjacent bit time-spans.

Given the Lorentzian response for a single transition, we can now construct the full Lorentzian model for the magnetic recording channel. Assume that our input data to be recorded is a sequence of values $b_n \in \{+1, -1\}$ such that our write current

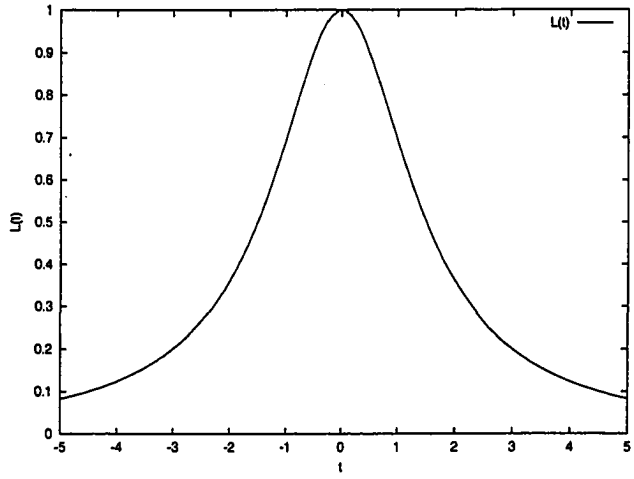


Figure 1.2: Lorentzian response for $S = 3$, $T = 1$.

is

$$i_{\text{write}}(t) = Ab_n \quad \text{if} \quad t \in [nT, (n+1)T) \quad \forall n \qquad . \tag{1.5}$$

Then the Lorentzian channel model says our read current will be

$$i_{\text{read}}(t) = \left(\sum_{n=-\infty}^{\infty} (b_n - b_{n-1}) L(t - nT) \right) + n(t) \tag{1.6}$$

where $n(t)$ is additive white Gaussian noise (AWGN), i.e., $n(t)$ is uncorrelated noise, normally distributed, with some variance σ^2. (Note that the $b_n - b_{n-1}$ term takes into account that the Lorentzian response is a response to *changes* in the magnetization.)

1.2. Sampling and Equalization

The output $i_{\text{read}}(t)$ is a continuous function of time t and, as such, is inconvenient to handle in modern digital signal processing systems; such systems really want to handle discrete-time signals sampled at regular intervals. Thus, given our $i_{\text{read}}(t)$, we would like to have a sequence of output values s_n, one for each bit time interval. We

would also like to mitigate, as far as possible, the effects of the inter-symbol interference. This mitigation is done by filtering the signal, thus changing its frequency content; this process is called **equalization**. We now describe how this sampling and equalization is done and give a model of a typical sampling and equalization system.

Our sampler/equalizer model is essentially the one given in [1]. The block diagram for the equalizer is shown in Fig. 1.3. The read current $i_{\text{read}}(t)$ is fed into a three-pole analog filter whose poles are specified by three parameters $a, b, c \in \mathbb{R}$. The impulse response of the analog filter is

$$p(t) = [e^{-(a+jb)t}u(t)] \star [e^{-(a-jb)t}u(t)] \star [e^{-ct}u(t)] \tag{1.7}$$

where $\star$ is the convolution operator and $u(t)$ is the unit step response

$$u(t) = \begin{cases} 1 & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases} . \tag{1.8}$$

Once the signal is analog filtered, it is sampled at discrete times $\tau + nT$ to get a discrete-time sequence s_n as follows:

$$s_n = [i_{\text{read}}(t) \star p(t)]|_{t=nT+\tau} . \tag{1.9}$$

The s_n sequence then passes through a digital filter with impulse response $f(D)$ to get an output sequence z_n. The parameters $a, b, c, \tau, f(D)$ of the sampler/equalizer system are all chosen to give a certain overall **target response** $g(D)$, i.e., so that for any randomly chosen input sequence b_n, the output z_n is close to b_n filtered by the response $g(D)$. That is, if

$$d(D) = g(D)b(D) \tag{1.10}$$

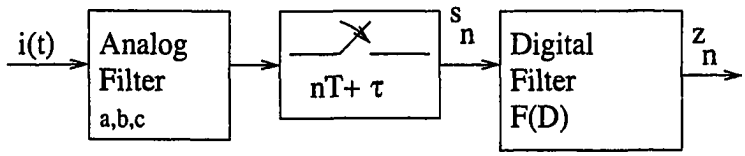


Figure 1.3: Diagram of sampler/equalizer system.

then z_n is approximately d_n. More precisely, the parameters are chosen to minimize the mean-squared error between z_n and the ideal $g(D)$-filtered output d_n.

The procedure to minimize this mean-squared error is somewhat complex, and proceeds as follows: suppose for the moment that we are given values of the parameters for the analog portion of the system (a, b, c, τ) as well as the target response coefficients

$$g(D) = \sum_{k=0}^{L} g_k D^k \tag{1.11}$$

and also suppose we have specified the size of our digital filter as $2M + 1$. We can find the coefficients of our digital filter

$$f(D) = \sum_{k=-M}^{M} f_k D^k \tag{1.12}$$

through a least-squares procedure. From (1.6) and (1.9) we can determine the values of the sampled sequence s_ℓ:

$$s_\ell = \sum_{k=-\infty}^{\infty} (b_{\ell-k} - b_{\ell-k-1})\gamma_k + n_\ell \tag{1.13}$$

where the γ_k coefficients are sampled versions of $L(t)$ convolved with $p(t)$:

$$\gamma_k = \int_{-\infty}^{\infty} L(t)p(kT + \tau - t)dt \tag{1.14}$$

and n_ℓ is noise, Gaussian but **not** white, having autocorrelation function

$$R_{nn}(D) = \sigma^2 \sum_{k=-\infty}^{\infty} r_k D^k \tag{1.15}$$

where

$$r_k = \int_{-\infty}^{\infty} p(t)p(t+kT)dt \quad . \tag{1.16}$$

Note that the γ_k and r_k are functions of the analog parameters a, b, c, τ. Anyway, given s_ℓ, we have the digital filter output

$$z_m = \sum_{\ell=-M}^{M} f_\ell s_{m-\ell} = \sum_{\ell=-M}^{M} f_\ell \left(n_{m-\ell} + \sum_{k=-\infty}^{\infty} (b_{m-\ell-k} - b_{m-\ell-k-1})\gamma_k \right) \tag{1.17}$$

and thus the error term

$$e_m = z_m - d_m = \sum_{\ell=-M}^{M} f_\ell \left(n_{m-\ell} + \sum_{k=-\infty}^{\infty} (b_{m-\ell-k} - b_{m-\ell-k-1})\gamma_k \right) - \sum_{n=0}^{L} g_n b_{m-n} \quad . \tag{1.18}$$

We want to minimize the average squared error

$$J = E[e_n^2] \quad . \tag{1.19}$$

We assume that the input data sequences are uncorrelated, so

$$E[b_i b_j] = \delta_{ij} \tag{1.20}$$

($E[b_i^2] = 1$ since b_i is either $+1$ or -1). Given this and the known autocorrelations of the filtered noise n_ℓ

$$E[n_i n_j] = \sigma^2 r_{i-j} \tag{1.21}$$

the expression for J simplifies remarkably to

$$\begin{aligned} J &= \sum_{\ell=-M}^{M}\sum_{j=-M}^{M} f_\ell f_j \sum_{k=-\infty}^{\infty} \gamma_k(2\gamma_{k-j+\ell} - \gamma_{k-j+\ell-1} - \gamma_{k-j+\ell+1}) \\ &\quad +\sigma^2 \sum_{\ell=-M}^{M}\sum_{j=-M}^{M} f_\ell f_j r_{\ell-j} \\ &\quad -2\sum_{\ell=-M}^{M}\sum_{p=0}^{L} f_\ell g_p(\gamma_{p-\ell} - \gamma_{p-\ell-1}) \\ &\quad +\sum_{n=0}^{L} g_n^2 \qquad . \end{aligned} \tag{1.22}$$

We can rephrase this more conveniently in matrix form by defining the following matrices:

$$\begin{aligned} \mathbf{f} &= [f_{-M}, f_{-M+1}, \ldots, f_M]^T \\ \mathbf{g} &= [g_0, \ldots, g_L] \\ \mathbf{T} &= [\gamma_{i-j} - \gamma_{i-j-1}]_{i\in[-M,\ldots,M],j\in[0,\ldots,L]} \\ \mathbf{R} &= [\sigma^2 r_{i-j} + \sum_{k=-\infty}^{\infty} \gamma_{k-i}(2\gamma_{k-j} - \gamma_{k-j-1} - \gamma_{k-j+1})]_{i,j\in[-M,\ldots,M]} \end{aligned} \tag{1.23}$$

and using these rewrite (1.22) as

$$J = \mathbf{f}^T\mathbf{R}\mathbf{f} - 2\mathbf{f}^T\mathbf{T}\mathbf{g} + \mathbf{g}^T\mathbf{g} \qquad . \tag{1.24}$$

Now the least-squares solution for $\mathbf{f}$ is readily derivable as

$$\hat{\mathbf{f}} = \mathbf{R}^{-1}\mathbf{T}\mathbf{g} \tag{1.25}$$

and the value of the error at this minimum is

$$J_{min} = \mathbf{g}^T\mathbf{g} - \hat{\mathbf{f}}^T\mathbf{T}\mathbf{g} \qquad . \tag{1.26}$$

(Note that the presentation in [1] differs somewhat from what we show here; in [1] the author dealt with an extended version of the Lorentzian channel model, and also the author imposed an, in our opinion arbitrary, restriction that $g(D)$ must be monic, making the analysis more complicated than it really needed to be.)

We now know how to minimize the mean-squared error J if we are given the analog subsystem parameters a, b, c, τ. To find the overall minimum, we have to find a, b, c, τ that lead to a minimal J_{min}. We do this as follows:

- Let τ take on values that are multiples of $0.1T$ from $-0.5T$ to $3T$. (For practical reasons it is not really possible to specify τ more precisely than about one-tenth the bit time T, so we need only consider τ values at these discrete intervals. The above range of possible τ values is somewhat arbitrary, but seems to work well in practice.)

- For each of these τ values, consider the preceding least-squares solution (1.25), (1.26) as a function mapping a,b,c,τ values to values of the least-squares minimum error J_{min}. Do a gradient search on the a, b, c parameter space to find the values of a, b, c that minimize J_{min}.

- Once this is done for all our τ values, select the τ that gave the best value and use the corresponding a, b, c parameters and the corresponding optimal $\hat{\mathbf{f}}$ filter parameters to specify our equalizer.

1.3. The Choice of Target Response $g(D)$

Given that the goal of the equalizer is to eliminate ISI, the obvious choice of target $g(D)$ would be $g(D) = 1$, which gives an ideal equalizer output

$$d_m = \sum_{n=0}^{0} g_n b_{m-n} = b_n \tag{1.27}$$

equal to the original data input, so the ISI is completely gone. This is an obvious choice, but one that does not work very well in practice. The problem is a phenomenon called **noise enhancement**. The equalizer is a linear filter designed to compensate for the ISI of the Lorentzian channel by altering the overall frequency response of the system. The problem is that the equalizer also changes the noise, increasing the noise power in proportion to the amount by which we are changing the frequency response. The case of $g(D) = 1$ produces a large change in frequency response, so our equalizer produces a signal that has no ISI, but has a lot of noise.

Let us examine this issue of noise enhancement more closely. Consider our Lorentzian channel model (1.6) and look at it as a linear system, with discrete-time inputs b_n and continuous-time output $i_{\text{read}}(t)$. For an input which is a delta function $b_n = \delta_{n0}$ the system has an output

$$i_{\text{read}}(t) = L(t) - L(t - T) \tag{1.28}$$

(we are for the moment ignoring the noise term) and hence the mapping $b_n \rightarrow i_{\text{read}}(t)$ is a linear filter with frequency response

$$\mathcal{F}[i_{\text{read}}(t)] = I(j\omega) = \hat{L}(j\omega)(1 - \exp(-j\omega t)) \tag{1.29}$$

where

$$\hat{L}(j\omega) = \frac{ST\pi}{2} \exp(\frac{-ST|\omega|}{2}) \quad . \tag{1.30}$$

(Obviously, the sequence $b_n = \delta_{n0}$ cannot occur as an actual input to the magnetic recording channel in real life, since the b_n values must be $+1$ or -1, but we are for the moment considering the Lorentzian channel model as an abstract linear system.)

Now, our equalizer is a combination of

- an analog filter with frequency response $P(j\omega) = \mathcal{F}[p(t)]$,

- a sampler, which in the frequency domain corresponds to a convolution of the Fourier transform of the signal with the function

$$X(j\omega) = \frac{2\pi}{T}\exp(-j\omega\tau)\sum_{n=-\infty}^{\infty}\delta(\omega - \frac{2\pi n}{T}) \tag{1.31}$$

and

- a digital filter, with frequency response

$$F(j\omega) = \sum_{k=-M}^{M} f_k \exp(-j\omega kT) \quad . \tag{1.32}$$

We know that the equalizer combined with the Lorentzian channel model gives a system whose overall frequency response is

$$G(j\omega) = \sum_{k=0}^{L} g_k \exp(-j\omega kT) \tag{1.33}$$

so we must have that

$$G(j\omega) = F(j\omega)((I(j\omega)P(j\omega)) \star X(j\omega)) \quad . \tag{1.34}$$

We would like a simple expression for $G(j\omega)$ as some function times $I(j\omega)$, but we do not quite have that, because of the complicating effect of the sampling function $X(j\omega)$. However, we can make an approximation that lets us ignore $X(j\omega)$, since the effect of sampling is to sum our frequency-domain response with several shifted copies of itself. Given moderately large channel densities (say $S = 3$), and given that $P(j\omega)$ is known to be a low-pass filter, the shifted copies of $I(j\omega)P(j\omega)$ should not overlap too much, so we should not see aliasing. And since we are, at the end, interested in $G(j\omega)$ only within the frequency range $|\omega| < \pi/T$ (since $G(j\omega)$ repeats itself outside that range), we can dispense with the $X(j\omega)$ convolution altogether.

We can thus subsume $P(j\omega)$ and $F(j\omega)$ into an overall equalizer response as follows:

$$G(j\omega) \approx H(j\omega)I(j\omega)\frac{2\pi}{T} \tag{1.35}$$

and hence

$$H(j\omega) \approx \frac{G(j\omega)}{I(j\omega)2\pi/T} \quad . \tag{1.36}$$

Now the signal-to-noise ratio (SNR) at the input to the equalizer is the ratio of the "signal" power $\int |L(t) - L(t-T)|^2 dt$ to the noise power, i.e.,

$$\begin{aligned} \text{SNR}_{in} &= \frac{\int_{-\infty}^{\infty} |I(j\omega)|^2 d\omega}{\sigma^2} \\ &\approx \frac{\int_{-\pi/T}^{\pi/T} |I(j\omega)|^2 d\omega}{\sigma^2} \quad . \end{aligned} \tag{1.37}$$

The signal power at the output is just

$$\sum_{k=0}^{L} g_k^2 = \frac{T}{2\pi} \int_{-\pi/T}^{\pi/T} |G(j\omega)|^2 d\omega \tag{1.38}$$

and the noise at the output is just white noise filtered by $H(j\omega)$, so we can compute the SNR at the output as

$$\text{SNR}_{out} = \frac{\sum_{k=0}^{L} g_k^2}{\int_{-\pi/T}^{\pi/T} |H(j\omega)|^2 \sigma^2 d\omega} \tag{1.39}$$

and hence

$$\frac{\text{SNR}_{in}}{\text{SNR}_{out}} \approx \frac{\left(\int_{-\pi/T}^{\pi/T} |I(j\omega)|^2 d\omega\right)\left(\int_{-\pi/T}^{\pi/T} |H(j\omega)|^2 d\omega\right)}{\sum_{k=0}^{L} g_k^2} \tag{1.40}$$

or, alternately,

$$\frac{\text{SNR}_{in}}{\text{SNR}_{out}} \approx \frac{\left(\int_{-\pi/T}^{\pi/T} |I(j\omega)|^2 d\omega\right)\left(\int_{-\pi/T}^{\pi/T} |H(j\omega)|^2 d\omega\right)}{\frac{T}{2\pi}\int_{-\pi/T}^{\pi/T} |G(j\omega)|^2 d\omega} \quad . \tag{1.41}$$

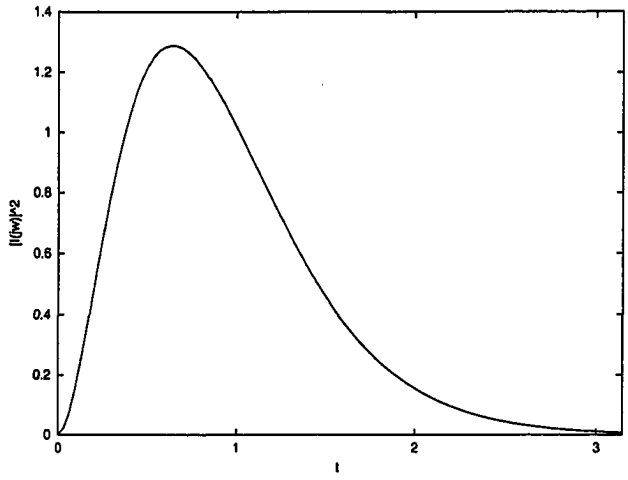


Figure 1.4: Plot of Lorentzian pulse response spectral density for S=3.

This ratio gives us a measure of the noise enhancement. Note that if $I(j\omega)$ was equal in magnitude to $G(j\omega)$, we would have a flat equalizer response $|H(j\omega)| = \frac{T}{2\pi}$, and the above ratio would be 1.

A plot of the power spectral density of the Lorentzian pulse response $|I(j\omega)|^2$ is shown in Fig. 1.4 for $S = 3$. Given that the spectrum for the target $G(D) = 1$ (no ISI) is flat, and given the shape of the $|I(j\omega)|^2$, it is clear that the noise enhancement for this target will be quite high. Practical systems for magnetic recording typically employ instead target responses that are short in length (thus limiting the ISI), but still have spectra closer in shape to that of $|I(j\omega)|^2$. In particular, they will often have spectral nulls at $\omega = 0$. Typical examples of these responses include

- the **Dicode** target

$$g(D) = 1 - D \tag{1.42}$$

  (note the spectral null at $\omega = 0$),

- the **Partial Response Four** (PR4) target

$$g(D) = (1 - D)(1 + D) \tag{1.43}$$

(note this has a null at $\omega = 0$ and at $\omega = \pi/T$),

- the **Extended Partial Response Four (EPR4)** target:

$$g(D) = (1-D)(1+D)^2 = 1 + D - D^2 - D^3 \quad , \tag{1.44}$$

- the **Modified Extended EPR4 (MEEPR4)** target [2]:

$$g(D) = 5 + 4D - 3D^2 - 4D^3 - 2D^4 \quad . \tag{1.45}$$

Table 1.1 shows the values of the noise enhancement ratio $\frac{\text{SNR}_{in}}{\text{SNR}_{out}}$ for these targets at various values of the channel density S. Note that for some of the higher-length channels we actually get negative noise enhancement, i.e. SNR_{out} is better than SNR_{in}. This is not too surprising; consider what would happen if $G(j\omega)$ was a very narrow-band bandpass response with its peak near that of $I(j\omega)$, and hence $H(j\omega)$ was a very narrow-band bandpass filter. In that case, $H(j\omega)$ would reject most of the noise while still letting the strongest portion of the signal through. However, the problem with such a narrow-band response is that narrow frequency response $G(j\omega)$ automatically implies a long-lasting time-domain response $g(D)$ (i.e., many non-zero terms in $g(D)$). This fits well with the results in Table 1.1, where the responses with least noise enhancement are also the ones with the longest time-domain response. Hence, our goal of minimizing noise enhancement conflicts with our goal of limiting the length of the ISI.

We assumed in this analysis a perfect filter, i.e., that $H(j\omega)$ was able to perfectly match the frequency response needed to turn the Lorentzian response into $G(j\omega)$. In practice, that is not the case; instead, we have some sort of equalizer with a limited set of parameters, and we try to find the set of parameters that provide a response as close as possible to the desired target, minimizing some parameter as in (1.22),

Table 1.1: Noise enhancement (in dB) versus channel density for various targets.

$g(D)$	$S = 2.4$	$S = 2.6$	$S = 2.8$	$S = 3.0$	$S = 3.2$
1	28.6791	28.1001	27.5561	27.0493	26.5846
$1 - D$	6.8670	8.3136	9.8402	11.4389	13.1022
$(1 - D)(1 + D)$	1.5512	2.4086	3.3757	4.4428	5.6010
$1 + D - D^2 - D^3$	-0.6226	-0.1715	0.3931	1.0638	1.8330
$5 + 4D - 3D^2 - 4D^3 - 2D^4$	-1.1846	-0.8665	-0.4059	0.1939	0.9285

(1.25). This results in a noise enhancement value different in practice from the one computed here. Furthermore, since the output of the real equalizer is not exactly the ideal output (the input filtered by the ideal $g(D)$), we have an error term which can be thought of as an additional noise source on the output, lowering SNR_{out} and thus acting as another source of noise enhancement. (Strictly speaking, this error term is not an uncorrelated noise source, since the error term is correlated with the inputs b_m, but it simplifies the analysis considerably to think of the error term as added white noise.)

The reasons for selecting one target over another may include other reasons than just how well the target response spectrum matches that of the Lorentzian channel. For example, Nishiya et al. [2] chose their MEEPR4 target not just because it provided a particularly good fit to the Lorentzian channel's spectrum (which it does), but also because they studied the most probable errors that would occur due to noise and devised a so-called maximum-transition-run (MTR) code that would have the advantage of suppressing some of these error sequences.

1.4. Magnetic Recording Systems

Now we are ready to consider what a simple system for magnetic recording of data might look like. A block diagram of such a system is shown in Fig. 1.6. The input data bits are mapped to $+1$ and -1 values, making a bipolar sequence. These b_m

values are then passed into the magnetic recording channel (MRC). The output of the channel is then equalized, as we discussed in the previous section. The output of the equalizer, z_m, is equal to the original b_m sequence filtered by the target response $g(D)$ and corrupted by noise. In order to get our original data back, we need some system that can look at the z_m and compute the original b_m. Since the z_m are noisy and thus we cannot know for sure what the b_m are, we must compute the b_m sequence that is most likely to have produced the observed z_m values. Such a system is called a **channel decoder**. The idea of using an MRC equalized to a particular target response and then doing maximum-likelihood decoding is called **partial-response maximum-likelihood** (PRML) decoding. The first PRML implementations did channel decoding using the Viterbi algorithm [3]. As we shall discuss in later portions of this dissertation, however, we will use the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm [4], because it provides not just the decoded sequence b_m but estimates of what the probabilities are that each bit b_m is a $+1$ or -1. If one layers an error-correcting code (ECC), such as the Gallager or LDPC codes [5], [6] on top of this simple PRML system, these probability estimates will be needed for the ECC decoder to function. Both the Viterbi and BCJR algorithms are based on considering the $g(D)$-equalized channel as a state machine that at each time takes in a bipolar input b_m (or, equivalently, a binary input x_m) and produces an output z_m. Each state of the state machine corresponds to a possible sequence of previous input values $b_{m-L}, \ldots, b_{m-1}$. Since each b_m can have one of two values, our state machine has 2^L possible states. As an example, we present the state transition diagram for EPR4 in Fig. 1.5. Each arrow is labeled with the input and output values for each transition, e.g., the arrow from state 1 to state 1 corresponds to an input $x_m = 1$ giving an output $z_m = 0$.

A diagram of a magnetic recording system using an LDPC code is shown in Fig. 1.7. This system has an LDPC encoder inserted in front of the magnetic

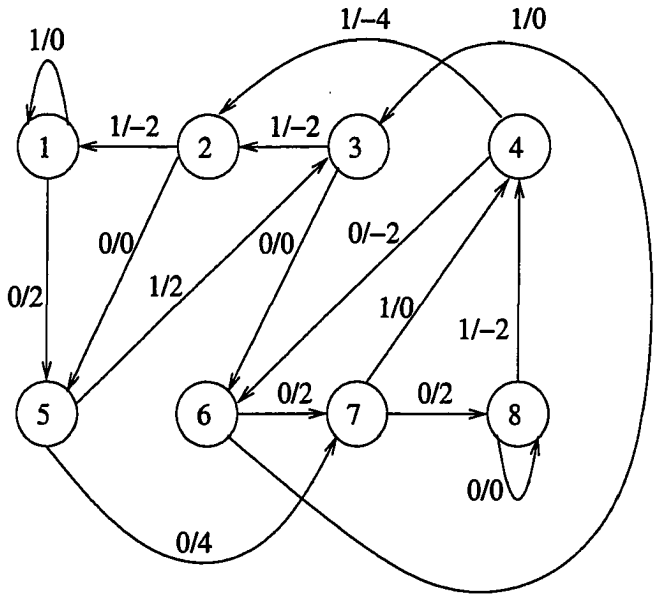


Figure 1.5: State transition diagram for EPR4 channel, mapping binary inputs to real outputs.

recording channel and an LDPC decoder inserted after the channel decoder. The LDPC decoder and the channel decoder both work on Bayesian principles, with each decoder computing *a posteriori* probabilities of each codeword bit x_m being either one or zero. The channel decoder computes *a posteriori* probabilities based on the *a priori* probabilities it takes in as well as the observed channel data z_m and the knowledge that z_m is a $g(D)$-filtered version of our bipolar sequence b_m. The LDPC decoder computes *a posteriori* probabilities based on its given *a priori* probabilities and the knowledge that the bits x_m must constitute a valid codeword of the LDPC code. Each decoder takes as *a priori* probabilities the *a posteriori* output of the other decoder, and the two decoders are run iteratively, one after the other. The initial channel decoder iteration starts with *a priori* probabilities $P(x_m = 1) = 0.5$, indicating no prior knowledge. This procedure of iterating the two decoders is called **turbo equalization** [7] in the literature. Once the iterations are done, the last set of probabilities computed can be used to hard decode the original x_m bits as follows:

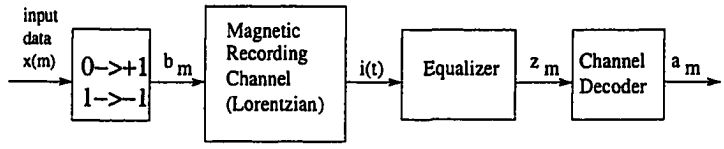


Figure 1.6: Diagram of simple system for PRML magnetic recording.

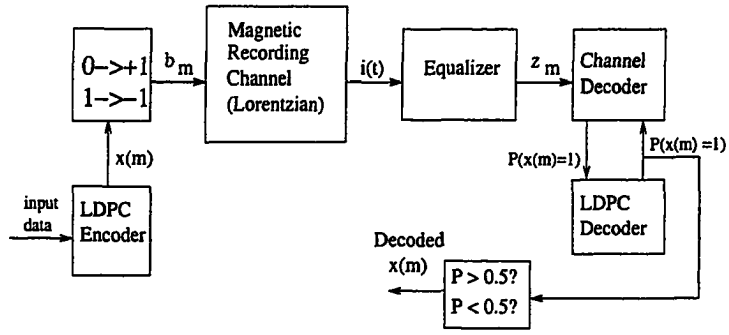


Figure 1.7: Diagram of simple system for PRML magnetic recording with LDPC error-correcting code.

we decide that bit x_m was a one bit if $P(x_m = 1) > 0.5$ and decide x_m was a zero bit otherwise. (Actually, in practice, one does a hard decoding of the probabilities after each iteration of turbo decoding, and checks to see if the resulting set of bits constitute a valid codeword. If they do, then the iteration is assumed to have converged to a valid result and subsequent iterations of turbo equalization may be skipped.)

1.5. Precoders and Maximum Transition Run Encoders

The previous section described simple PRML magnetic recording systems, either with or without an error-correcting code layer added. However, in practice, there are a couple of problems that require us to add a bit more complexity to our system.

The components we add are called **precoders** and **maximum transition run (MTR) encoders**, and we explain why they are needed and what they do below.

1.5.1. Precoders

Most partial response channels suffer from a problem called **quasi-catastrophic sequences** or **ambiguous output sequences**. (We have not seen the latter term used outside of [8], but we find it appropriate.) Let us consider the EPR4 state transition diagram in Fig. 1.5. Suppose we have four different input sequences of bits x_k about which we only specify the sequence of bits from time k on (i.e., specifying $x_k, x_{k+1}, \ldots$) and what the state of the channel is at time k, call this S_k. (Note that specifying S_k is equivalent to specifying x_{k-1}, x_{k-2} and x_{k-3}.) Table 1.2 gives our set of four sequences. A bit of careful study of Fig. 1.5 will show that all four of these sequences give the exact same output sequence from time k on:

$$z_m = 0 \quad \forall m \geq k \qquad . \tag{1.46}$$

This makes it difficult for the channel decoder given such an output sequence to determine which of the possible input sequences actually was transmitted. We have, as the term "ambiguous output sequences" implies, an ambiguity which is difficult to resolve, and this impairs the performance of the channel decoder.

To avoid the problem of ambiguous output sequences, we add an additional transformation step, a **precoder** on our binary data x_m just before we convert it to bipolar form. Precoders are basically infinite-impulse-response (IIR) digital filters, except that the digital filters operate in the binary field $\mathbb{Z}_2$ and not the field of real numbers. The precoder is characterized by a monic polynomial

$$p(D) = \bigoplus_{i=0}^{P} p_i D^i \qquad p_i \in \mathbb{Z}_2, p_0 = 1 \tag{1.47}$$

Table 1.2: Four quasi-catastrophic sequences for the EPR4 channel.

S_k	$(x_k, x_{k+1}, \dots)$
1	$1,1,1,1,\dots$
8	$0,0,0,0,\dots$
6	$1,0,1,0,\dots$
3	$0,1,0,1,\dots$

and the IIR filter is just $1/p(D)$. Hence, the precoder maps an input sequence of bits x_m to bits $\hat{x}_m$ where

$$\hat{x}_m = x_m \oplus \bigoplus_{i=1}^{P} p_i \hat{x}_{m-i} \quad . \tag{1.48}$$

Note that if this were a standard IIR filter we would have $x_m - \sum p_i \hat{x}_{m-i}$ on the right-hand side above, but since the precoder operates inside the field $\mathbb{Z}_2$, subtraction and addition are interchangeable. Actually figuring out what precoder $p(D)$ is needed and showing that it works is a somewhat tricky endeavor which we will not go into here; interested readers are referred to the Appendix of [8], which shows in detail that, e.g., $p(D) = 1 \oplus D^2$ produces a valid precoder for EPR4.

The reader may be wondering if we need another system on the output side of the magnetic recording channel to compensate for or undo the effects of the precoder. It turns out that we do not need a separate system to undo the precoding. The channel decoder itself is perfectly capable of doing this; all we have to do is use, instead of the state transition diagram of the original channel, an altered state transition diagram that contains the effects of both channel and precoder. In other words, we have a single state transition diagram that covers the mapping from un-precoded data x_m to channel outputs z_m, and we use that diagram to construct our channel decoder. As an example, Fig. 1.8 shows the combined diagram for EPR4 with the $1/(1 \oplus D^2)$ precoder.

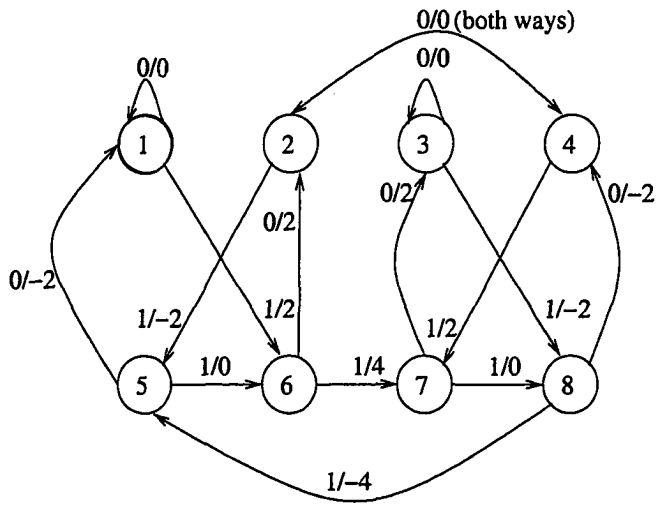


Figure 1.8: State transition diagram for precoded EPR4 channel, mapping binary inputs to real outputs.

1.5.2. MTR Encoders

Throughout this chapter, we have assumed that our magnetic recording system exhibits perfect synchronization. That is to say, if the writing process put bit x_m into the magnetic recording system at time m, then the reading process manages to get the corresponding channel output z_m at time m and not, say, one bit-time ahead or behind, or worse yet a fraction of a bit-time off. In practice, any magnetic recording system needs to have some system for ensuring that the reading process does not get out of sync. Designing systems to ensure synchronization is a subject of extensive research, and one we will not go into here; for the remainder of this work we shall continue to assume perfect synchronization. However, the synchronization systems that are commonly used do impose some constraints on the bipolar sequence b_m that we must allow for in our system design. The synchronization system usually uses some sort of phase-locked loop triggered by the transitions seen in the magnetic recording channel. This means that if there is a long sequence where there are no transitions, where $b_m = +1$ for a long time or $b_m = -1$ for a long time, the

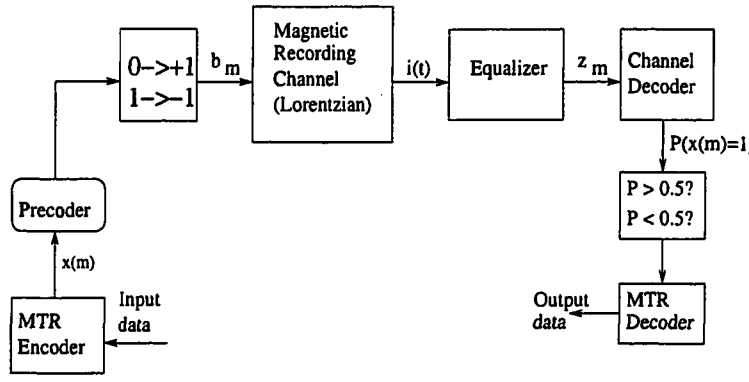


Figure 1.9: Diagram of simple system for PRML magnetic recording with precoder and MTR code.

synchronization system gets confused and loses sync. Hence, we must have an additional code to eliminate b_m sequences with such long runs without transitions. Such codes are called **run-length limited (RLL)** codes or **maximum-transition run (MTR)** codes; [2] discusses the design of such an MTR code. Fig. 1.9 shows a block diagram of a magnetic recording system with a precoder and with MTR code in use, but without the LDPC code. (Designing a system with both LDPC and MTR codes in use has some complications, which we discuss in the next subsection.) Note that since the MTR code must try to avoid certain sequences of non-transitions, but the MTR encoder is placed before the precoder, the MTR code **must** be designed with the specific precoder in mind.

There are other constraints which the MTR code must obey as well. Not only do we wish to avoid long runs with no transitions (because they break the synchronization systems), but we wish to avoid runs with lots of transitions right next to each other. Back in Section 1.1 we said that the magnetic recording channel was basically a linear channel, with one Lorentzian pulse response being super-imposed on the output for each transition in the input. It turns out that this is only approximately true; the responses from adjacent transitions interfere with each other in a non-linear way. The non-linearities are usually small enough to be safely ignored,

but not when there are a group of several transitions adjacent to each other; in that case, the Lorentzian channel model breaks down, and our recording systems, designed with that model in mind, will not work well. So we wish to avoid not only long sequences without transitions, but long sequences of nothing but transitions as well. The particular MTR code presented in [2] is designed to avoid runs of more than three transitions in a row, or more than eleven bit times without a transition; such a constraint is called an MTR(3;11) constraint.

1.5.3. Systems with both LDPC and MTR Encoders

We are now ready to consider systems which employ both LDPC error-correcting codes and MTR codes. However, it is not at first clear what order the various encoders should go in. Fig. 1.10 shows the overall system design we are using, a design originated by H. Song [9]. The reasoning behind this particular layout is as follows:

- There is no known soft-input/soft-output decoder for the MTR code. Soft-input/soft-output means that the decoder takes as its input *a priori* probabilities and gives *a posteriori* probabilities as its output, like the LDPC and channel decoders do. The MTR decoder instead takes in an MTR-encoded bit sequence and gives the original bit sequence. Hence the MTR decoder has to come **after** the LDPC decoder and the box that does the check to see if $P(x) > 0.5$ or not. This implies that the codes on the input have to go in the reverse order. The MTR encoder has to go **before** the LDPC encoder.

- We can do this, but we now have a problem. We have an MTR-encoded word that goes into the LDPC encoder. The LDPC encoder adds a group of parity bits to this codeword. The problem is that there is no guarantee that the parity bits obey the MTR constraint. To solve this problem, we have to

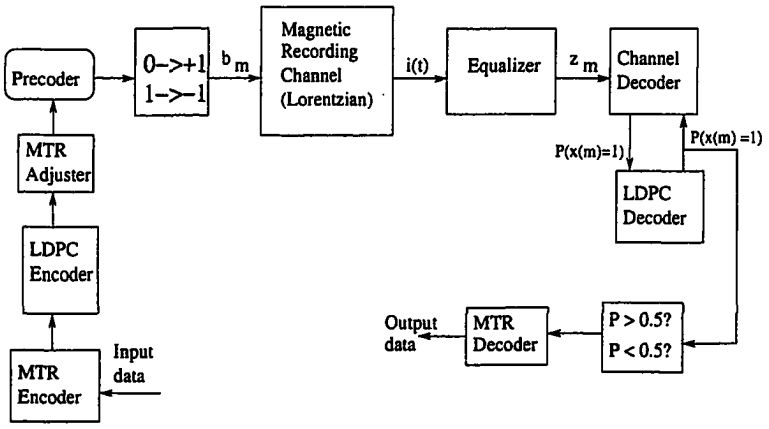


Figure 1.10: Diagram of system for PRML magnetic recording with precoder, MTR code, and LDPC code.

insert additional bits amongst the parity bits to ensure that we never violate the MTR constraint. We call the system that does this insertion the MTR adjuster. The resulting word looks something like Fig. 1.11. The bits added by the MTR adjuster after every three parity bits obey the following rule:

- If the preceding three bits are all ones, make this bit a zero.
- If the preceding eleven bits are all zeros, make this bit a one.
- Otherwise, the bit we add will not affect the MTR constraint, so it does not matter what value it has; we arbitrarily choose a zero bit.

These rules make the MTR adjuster's output obey the MTR(3;11) constraint. (Note that the MTR(3;11) code from [2] is made to work with a $1/(1 \oplus D)$ precoder, so zero bits always correspond to no transition and one bits correspond to transitions.)

LDPC Encoder Input

LDPC Parity Bit

Bits added by MTR adjuster

Figure 1.11: Output of the MTR constraint adjuster.

1.6. Sector Sizes: A Note

Much preceding discussion of the magnetic recording channel, and of basic magnetic recording systems such as Fig. 1.6, assumed implicitly that our data sequences were infinite in length, or at least of no fixed specific length. In practice this is not the case. In practical disk drives, we always read or write data in segments, called **sectors** of a fixed size, and the sectors are stored on separate regions of the magnetic medium. This allows us to write and rewrite one particular sector of data without disturbing other sectors elsewhere on the disk. The sectors are separated by blank (unwritten) regions so as to keep the ISI from one sector from affecting another, and also by regions of specially written data called **preambles** and **postambles**. The preambles and postambles come before and after the sector, respectively, and contain sequences of data designed to allow the synchronization systems to acquire sync on a known sequences of transitions. The preambles also contain data telling the number of each sector, helping our system to know which sector is currently passing by the magnetic recording head at the time. As with the workings of the bit-time synchronizers, we are not interested here in exact details of what the preambles and postambles are like; for our purposes, it is enough to know that our magnetic recording system deals in bits in sector-size segments. A sector is usually 4096 bits long in current disk drives; future disk drives may have

larger sectors, and exactly how much larger the sectors should be is a matter of current discussion [10]. The sector size is an important consideration in designing our LDPC code as well as the BCJR decoder. Unlike the Viterbi decoder used in the earliest PRML recording systems, which did not care about block lengths and operated in a streaming fashion, both the LDPC and BCJR algorithm are block algorithms, operating on a block of data at a time. It is convenient for the block size of these algorithms to be big enough to hold a sector. Hence, in the system of Fig. 1.7, the LDPC code must have 4096 data bits and thus must have a codeword length of $4096/R$ bits, where R is the code rate. Since we want to fit as much user data as possible on our magnetic medium, we want a high code rate, so typically R will be 0.9 or higher. The BCJR decoder's block length must be large enough to handle the LDPC codeword, so it must also be $4096/R$, though often we may add a few "trailer" bits, extra bits that we add to the end of our codeword to force the channel back to a known starting state. This trailer helps the performance of the BCJR algorithm a little. For the system described in Fig. 1.10, the additional MTR layer adds 1/16 times as many bits, and the MTR adjuster adds some bits too. Specifically, the number of LDPC data bits in this case is

$$K = 4096\frac{17}{16} = 4352 \tag{1.49}$$

and the number of parity bits added is

$$L = K\left(\frac{1}{R} - 1\right) = 4352\left(\frac{1-R}{R}\right) \tag{1.50}$$

and the number of bits added by the MTR adjuster is $L/3$, for a total of

$$\begin{aligned} N_{BCJR} &= K + L + \frac{L}{3} \\ &= 4352\left(1 + \frac{1-R}{R} + \frac{1-R}{3R}\right) \\ &= 4352\frac{4-R}{3R} \end{aligned} \tag{1.51}$$

bits as the block length for the BCJR decoder.

Chapter 2

The BCJR Algorithm

As we mentioned in the previous chapter, the BCJR algorithm is the algorithm we use to handle the effects of the ISI induced by the channel. As we discussed previously, and as shown in Fig. 2.1, bits of data $x(t)$ go into the channel producing outputs $z(t)$. The BCJR decoder takes the channel outputs $z(t)$ as well as values specifying the *a priori* probability that each bit is a zero or one bit, and produces *a posteriori* probabilities for each of those bits. In this chapter we explain how exactly the BCJR algorithm works.

2.1. Log-Likelihood Values

In the BCJR algorithm, we find it convenient to use somewhat modified forms of the probabilities $P(x(t) = 0)$ or $P(x(t) = 1)$. Instead of dealing with these probabilities explicitly, we use what are called **log-likelihood ratios**

$$L(x(t)) = \log \frac{P(x(t) = 0)}{P(x(t) = 1)} \quad . \tag{2.1}$$

A log-likelihood ratio contains the same information as either of the probabilities $P(x(t) = 0)$ or $P(x(t) = 1)$; we can freely convert from probabilities to log-likelihood ratios or vice versa:

$$\begin{aligned} P(x(t) = 1) &= \frac{\exp(-L(x(t))/2)}{\exp(L(x(t))/2) + \exp(-L(x(t))/2)} \\ P(x(t) = 0) &= \frac{\exp(L(x(t))/2)}{\exp(L(x(t))/2) + \exp(-L(x(t))/2)} \end{aligned} \tag{2.2}$$

(note that $P(x(t) = 0) + P(x(t) = 1) = 1$, as it should). Positive values of $L(x(t))$ correspond to $x(t)$ being more likely to be a zero bit, and negative values of $L(x(t))$ correspond to $x(t)$ being more likely to be a one bit. $L(x(t)) = 0$ corresponds to $x(t)$ being equally likely to be a one or zero bit, or to us being totally ignorant of the state of $x(t)$. In the initial round of BCJR decoding, we start with *a priori*

probabilities of 0.5, so our initial L values are all zero.

We find log-likelihoods convenient numerically because for bits whose value we are particularly certain about, we risk having numerical errors if use the probabilities and not the log-likelihoods. Suppose that for bit $x(t)$ we have $P(x(t) = 0) = 10^{-30}$, then on a typical 16-decimal-place accuracy computer we would have $P(x(t) = 1) = 1.0000000000000000$. Just looking at $P(x(t) = 1)$, we would have no idea whether the probability of that bit being zero was 10^{-30} or 10^{-40}; both would give $P(x(t) = 1) = 1.0000000000000000$. We could alternatively choose to keep track of only $P(x(t) = 0)$ values and not $P(x(t) = 1)$ values (since the two, theoretically, contain the same information about $x(t)$), but we would run into similar problems with bits we are highly certain are zeros. We could keep track of $P(x(t) = 0)$ and $P(x(t) = 1)$ values separately throughout our calculations, but that would take twice as much memory and might still cause numerical problems. But if we do everything in the log-likelihood domain, everything is nicer; we just have a single number $L(x(t))$, and we can easily tell the difference between $P(x(t) = 0) = 10^{-30}$ or 10^{-40} because $L(x(t))$ will be +69.078 in one case and +92.013 in the other. For this reason, while the computations in the BCJR algorithm can be done either with probabilities or log-likelihoods, we prefer to use the log-likelihood values. Hence we will present in this chapter the so-called **log-domain** version of the BCJR algorithm, and only occasionally make note of how the non-log version of the BCJR differs. For the full details of the non-log version readers may refer to [11]. (We shall see in the next chapter that the LDPC decoder algorithm also comes in both log and non-log versions.) The choice of convention in defining the log-likelihoods in (2.1) is completely arbitrary; we could equally well have chosen

$$L(x(t)) = \log \frac{P(x(t) = 1)}{P(x(t) = 0)} \quad . \tag{2.3}$$

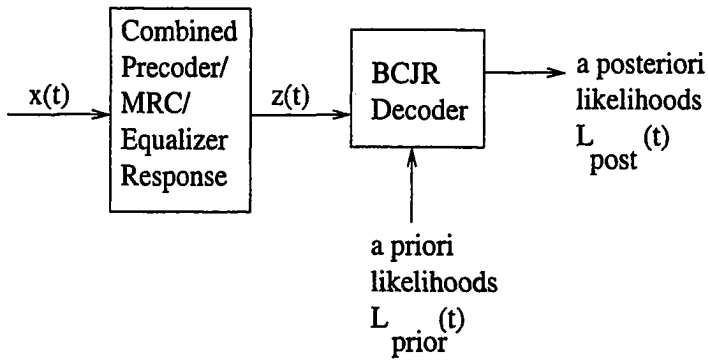


Figure 2.1: The BCJR Decoder.

as a convention. We note that while the literature we have seen about the BCJR algorithm tends to employ one convention for defining $L(x(t))$, papers about LDPC decoding tend to employ the other convention; this is a bit confusing for people like us who need to use both algorithms at once. We choose to use the convention in (2.1) throughout this work to be self-consistent. A further notational note; since in the BCJR algorithm we only deal with likelihoods of the bits $x(t)$ and never need likelihoods of any other random variables, we simplify the notation $L(x(t))$ to just $L(t)$.

2.2. Basic Operation of the BCJR Algorithm

Here we follow the presentation of the log form of the BCJR algorithm more or less as in [12]. Let us define some notation. Our channel (and any precoder, if present) can be described by a state-transition diagram such as Fig. 1.5. At each time t the channel receives a binary input $x(t)$ and produces an output $z(t) \in \mathbb{R}$ which is equal to the ideal state-machine output $\hat{z}(t)$ corrupted by additive white Gaussian noise. The mapping between $x(t)$ and $\hat{z}(t)$ depends on the current **channel state** s_t, which is for convenience represented as an integer in $[0, M-1]$, where M is the number of states. The state at time $t+1$ is also a function of the current state and

channel input, so the channel can be fully described by giving the AWGN variance σ^2 and writing down a table of the function

$$f : (s(t), x(t)) \to (s(t+1), \hat{z}(t)) \quad . \tag{2.4}$$

This is just a way of writing as a function the various transitions in our state-transition diagram.

For the BCJR algorithm, it is more convenient to rewrite this function in terms of all possible pairs of states $(s(t), s(t+1))$ that can begin and end at a given time-step. Let D be the set of all such state pairs that actually can occur in the channel. Then we can define two new functions $I(s(t), s(t+1))$ and $O(s(t), s(t+1))$ for all $(s(t), s(t+1)) \in D$ such that

$$I(s(t), s(t+1)) = x(t) \quad \text{and} \quad O(s(t), s(t+1)) = \hat{z}(t) \quad . \tag{2.5}$$

Note that it is not immediately obvious that these are well defined functions, i.e., that there is a unique input x for each state transition $(s(t), s(t+1))$, but in fact this is the case for PR channels with the states defined in the "obvious" way (where the state $s(t)$ is the bit vector encoding of the previous channel inputs $x(t-1), x(t-2), \ldots$). These functions are also well defined in the case of a precoded channel.

We assume that a word of length N bits is sent through the channel and that the resulting channel outputs $z(0), z(1), \ldots, z(N-1)$ are given to the BCJR decoder. The starting state s of the channel is assumed known, as is the ending state e. We assume the words always contain a trailing section such as to always leave the channel in state e. (If this assumption does not hold, a few minor modifications to the BCJR algorithm below are needed.) We also assume that for each time $t \in [0, N-1]$ we have an *a priori* log-likelihood ratio $L_{\text{in}}(t)$ computed from the *a priori* probabilities for bit $x(t)$.

The BCJR algorithm computes *a posteriori* likelihood values $L(t)$ for each time t. The procedure for computing them consists of three steps.

2.2.1. Step 1: Forward pass – compute m values

The first part of the algorithm goes forward in time and computes values $m_i(t)$ for each state i and time t. The $m_i(t)$ are (possibly rescaled) log versions of the probabilities that the channel is in state i at time t. (In the non-log version of the BCJR algorithm they are just the probabilities of being in state i at time t.)

We start by initializing the m values for $t = 0$:

$$m_s(0) = 0 \quad \text{and} \quad m_i(0) = -\infty \quad \forall i \neq s \qquad . \tag{2.6}$$

This corresponds to assuming that at time $t = 0$ we are certain to be in state s and not in any other state. Then for all $t \in [1, N-1]$ we successively compute

$$m_i(t) = \log \sum_{j:(j,i)\in D} \exp\left(m_j(t-1) + p(j,i,z(t-1), L_{\text{in}}(t-1)\right) \quad \forall i \tag{2.7}$$

where the function $p(j,i,z,L)$ is defined as:

$$p(j,i,z,L) = \begin{cases} -(z - O(j,i))^2/(2\sigma^2) + L/2 & \text{if } I(j,i) = 0 \\ -(z - O(j,i))^2/(2\sigma^2) - L/2 & \text{if } I(j,i) = 1 \end{cases} \qquad . \tag{2.8}$$

The $p(j,i,z,L)$ is basically a log version of the Gaussian probability density function of the noise being such as to give output z from the ideal channel output $O(j,i)$, plus the $\pm L/2$ terms which are a log version of the *a priori* probabilities. This is so

because

$$\log P_{\text{prior}}\left(x(t) = \left\{ \begin{array}{c} 0 \\ 1 \end{array} \right\}\right) = \pm\frac{L_{\text{in}}(t)}{2} - \log\left(\exp\left(\frac{L_{\text{in}}(t)}{2}\right) + \exp\left(-\frac{L_{\text{in}}(t)}{2}\right)\right) \tag{2.9}$$

and it turns out we can drop the second term because it will just disappear if we renormalize the $m_i(t)$, as we discuss below. (The above expression for $p(j,i,z,L)$ also omits a constant term $-1/2\log 2\pi\sigma^2$; this constant term also disappears in the renormalization.) Note that if $L = 0$ (as is the case for the initial round of BCJR decoding), $p(j,i,z,L)$ simplifies to

$$p(j,i,z,0) = -\frac{(z - O(j,i))^2}{2\sigma^2} \quad . \tag{2.10}$$

We should also note that none of the rest of the algorithm uses the $m_i(t)$ values in isolation; only differences between pairs of $m_i(t), m_j(t)$ values for differing i,j end up in the final result. Hence, the $m_i(t)$ values can be (and, where convenient, are) renormalized by adding or subtracting a constant value. In practical implementations of the BCJR algorithm, this is done to avoid numerical problems by keeping the $m_i(t)$ values from growing too large.

The non-log version of the update rules for $m_i(t)$ is essentially the same as (2.7), only with everything exponentiated to bring it the log domain, i.e., sums become products, the log-of-sum-of-exponentials combination becomes just a sum, etc.

Often, in practical implementations of the BCJR algorithm, the log and exponential operations are found to be too computationally expensive and one uses less expensive approximations to compute $m_i(t)$. One way is to use a piecewise-linear approximation to the log-sum-exponential function

$$G(x,y) = \log(\exp(x) + \exp(y)) \tag{2.11}$$

and use that in the computation of $m_i(t)$. An example of such an approximation is found in [12], where one computes

$$
\begin{aligned}
a &= \max(x, y) \\
b &= |a - b| \\
G(x, y) &\approx \begin{cases}
a + 0.6931 & \text{if} \quad b < 0.3571 \\
a + 0.3985 & \text{if} \quad 0.3571 \leq b < 1.0714 \\
a + 0.2148 & \text{if} \quad 1.0714 \leq b < 1.7857 \\
a + 0.1109 & \text{if} \quad 1.7857 \leq b < 2.5 \\
a + 0.0558 & \text{if} \quad 2,5 \leq b < 3.2143 \\
a + 0.0277 & \text{if} \quad 3.2143 \leq b < 3.9286 \\
a + 0.0137 & \text{if} \quad 3.9286 \leq b < 4.6429 \\
a + 0.0067 & \text{if} \quad 4.6429 \leq b < 5.3571 \\
a & \text{if} \quad b \geq 5.3571
\end{cases}
\end{aligned}
\tag{2.12}
$$

An even faster approximation can be made by noting that

$$G(x, y) \approx \max(x, y) \tag{2.13}$$

and hence

$$m_i(t) \approx \max_{j:(j,i)\in D} m_j(t-1) + p(j, i, z(t-1), L_{\text{in}}(t-1)) \quad \forall i \quad . \tag{2.14}$$

2.2.2. Step 2: Backward pass – compute $\bar{m}$ values

This step is essentially the same as Step 1, only working backwards from the end of the codeword to the beginning. We start by initializing the $\bar{m}$ values for $t = N$:

$$\bar{m}_e(N) = 0 \quad \text{and} \quad m_i(N) = -\infty \quad \forall i \neq e \quad . \tag{2.15}$$

This corresponds to asserting that the system at time N is certain to be in state e. If we do not employ the trailer method to ensure that our channel always ends in a known state e, we must instead set

$$\bar{m}_i(N) = 0 \qquad \forall i \tag{2.16}$$

corresponding to all states being equally probable. (Actually, equiprobable would mean each state would have probability $1/M$, so $\bar{m}_i(N) = \log 1/M$. But as we noted before, any constant common to all the $\bar{m}_i(t)$ can be made to disappear through renormalization.)

Then for $t = N-1, \ldots, 1$ we compute:

$$\bar{m}_i(t) = \log \sum_{j:(i,j)\in D} \exp\left(\bar{m}_j(t+1) + p(i,j,z(t),L_{\text{in}}(t))\right) \quad \forall i \qquad . \tag{2.17}$$

All the comments in the previous section regarding renormalization of the $m_i(t)$ and more efficient approximation techniques apply here to the $\bar{m}_i(t)$ as well.

2.2.3. Step 3: Compute L_0, L_1, and final L values

For each time $t = 0, 1, \ldots, N-1$, we compute two values $L_0(t)$ and $L_1(t)$ as follows:

$$\begin{aligned} L_0(t) &= \log \sum_{(i,j)\in D, I(i,j)=0} \exp\left(m_i(t) + p(i,j,z(t),L_{\text{in}}(t)) + \bar{m}_j(t+1)\right) \qquad (2.18) \\ L_1(t) &= \log \sum_{(i,j)\in D, I(i,j)=1} \exp\left(m_i(t) + p(i,j,z(t),L_{\text{in}}(t)) + \bar{m}_j(t+1)\right) \qquad . \end{aligned}$$

The $L_0(t)$ and $L_1(t)$ are measures of how likely each bit $x(t)$ is to be either a 0 or a 1, respectively. The final *a posteriori* likelihood value $L(t)$ is computed as

$$L(t) = L_0(t) - L_1(t) \quad . \tag{2.19}$$

One can see that if $L_0(t) = L_1(t)$, implying the BCJR decoder has equally good evidence that $x(t)$ is a zero or a one, then the *a posteriori* likelihood will be $L(t) = 0$, as we would expect. One can use approximations to the log-sum-exponential combination as before, e.g.,

$$L_0(t) \approx \max_{(i,j)\in D, I(i,j)=0} m_i(t) + p(i,j,z(t),L_{\text{in}}(t)) + \bar{m}_j(t+1) \tag{2.20}$$

and similarly

$$L_1(t) \approx \max_{(i,j)\in D, I(i,j)=1} m_i(t) + p(i,j,z(t),L_{\text{in}}(t)) + \bar{m}_j(t+1) \quad . \tag{2.21}$$

Chapter 3

Coding and Decoding of LDPC Codes

In this chapter, we discuss how to encode and decode codewords of a low-density parity-check code [5],[6]. We explain how to do decoding of such codes through the so-called **belief propagation** algorithm, also known as the **sum-product** algorithm. (There is another algorithm besides the belief propagation one that decodes LDPC codes [5], but as it is not a soft-input/soft-output algorithm, it is not of interest to us here.) As with the BCJR algorithm, there are both non-log and log versions of the belief propagation decoder; we shall explain both versions. We also discuss the **max-product** algorithm, a fast approximation to the full sum-product decoder, and discuss in a bit more detail how the LDPC and BCJR decoders feed information back and forth.

3.1. Basic Notation and Encoding

Each LDPC code is specified by two numbers K and L, the number of data bits and parity bits respectively, and a $L \times (K+L)$ parity-check matrix $\mathbf{H}$ of ones and zeros. $\mathbf{H}$ is usually a sparse matrix (hence the "low density" part of the LDPC name). The codewords produced by the LDPC encoder have length $N = K + L$, and each word $\mathbf{x} \in {\mathbb{Z}_2}^N$ is a valid LDPC codeword iff

$$\mathbf{H}\mathbf{x}^T = 0 \quad . \tag{3.1}$$

We commonly rearrange our $\mathbf{H}$ matrix to make the code a **systematic** code; such a code has the property that the codeword $\mathbf{x}$ consists of our original K data bits followed by L new bits, the parity bits. This simplifies encoding and decoding, as the encoder only has to add L new bits to the input data to make a codeword, and once one is done with finding the most likely codeword with the LDPC decoder, one can just throw away the final L bits to get the original data. Also, the MTR adjustment scheme mentioned in Section 1.5.3 would not work without a systematic

code.

Supposing we have $\mathbf{H}$ corresponding to a systematic LDPC code, we can rewrite $\mathbf{H}$ as the concatenation of two matrices as follows:

$$\mathbf{H} = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \end{array} \right] \tag{3.2}$$

where $\mathbf{A}$ has dimensions $L \times K$ and $\mathbf{B}$ has dimensions $L \times L$, and $\mathbf{B}$ is invertible. Let $\mathbf{x}_{\text{in}}$ be our input data word and $\mathbf{x}_{\text{p}}$ be our added parity bits, i.e.,

$$\mathbf{x} = \left[\begin{array}{c|c} \mathbf{x}_{\text{in}} & \mathbf{x}_{\text{p}} \end{array} \right] \quad . \tag{3.3}$$

Then our encoding procedure computes the parity bits like this:

$$\mathbf{x}_{\text{p}} = \left(\mathbf{B}^{-1}\mathbf{A}\mathbf{x}_{\text{in}}^T\right)^T \quad . \tag{3.4}$$

These parity bits produce a codeword that satisfies (3.1) since

$$\begin{aligned} \mathbf{H}\mathbf{x}^T &= \left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \end{array} \right] \left[\begin{array}{c|c} \mathbf{x}_{\text{in}} & \mathbf{x}_{\text{p}} \end{array} \right]^T \\ &= \left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \end{array} \right] \left[\begin{array}{c|c} \mathbf{x}_{\text{in}} & \left(\mathbf{B}^{-1}\mathbf{A}\mathbf{x}_{\text{in}}^T\right)^T \end{array} \right]^T \\ &= \mathbf{A}\mathbf{x}_{\text{in}}^T + \mathbf{B}\mathbf{B}^{-1}\mathbf{A}\mathbf{x}_{\text{in}}^T \\ &= \mathbf{A}\mathbf{x}_{\text{in}}^T + \mathbf{A}\mathbf{x}_{\text{in}}^T \\ &= 0 \end{aligned} \tag{3.5}$$

(remember we are dealing with vectors and arrays over $\mathbb{Z}_2$, so anything added to itself gives zero).

3.2. The Belief Propagation Decoder

We explain here how the belief propagation decoder operates, first with the non-log version of the algorithm, and later with the log version of the algorithm. Our favorite exposition of LDPC decoding through belief propagation is in an (unpublished) paper by Fan et al. [13]. We follow their notation here, deviating only where it seems appropriate, e.g., to maintain consistency with our notation elsewhere in this dissertation.

3.2.1. Message Passing over a Graph

The basic idea behind the belief propagation algorithm is to consider the LDPC code as a network of nodes and do Bayesian probability calculations on this network, propagating various probabilities (also called **beliefs** or **messages**) from one node to another throughout the calculation. Our network of nodes consists of two kinds of nodes. There are N **variable nodes**, each of which corresponds to a bit in the codeword $\mathbf{x}$. There are also L **check nodes**, each of which corresponds to one of the rows of $\mathbf{H}$, i.e., to a constraint equation that the codeword must satisfy:

$$\mathbf{H}_i\mathbf{x}^T = 0 \qquad i = 1, 2, \ldots, L \tag{3.6}$$

where $\mathbf{H}_i$ is the ith row of $\mathbf{H}$. Each variable node x_i is connected to the set of check nodes that the ith bit of $\mathbf{x}$ participates in, i.e., there are connections between the variable node x_i and all check nodes for rows j of $\mathbf{H}$ for which $\mathbf{H}_{ji} = 1$. An example of such a graph is shown in Fig. 3.1 for the parity-check matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \quad . \tag{3.7}$$

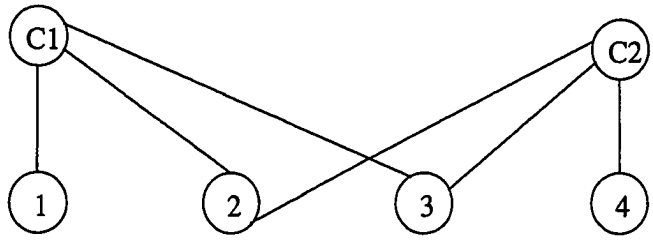


Figure 3.1: Network for a very simple LDPC code.

(This particular choice of $\mathbf{H}$ is not, strictly speaking, a low-density parity-check matrix, due to its small size and large number of ones, but it is a parity-check matrix. For practical reasons we did not wish to try and draw the network for a more realistic LDPC code here.) The messages passed from node to node are essentially estimates of *a posteriori* probabilities of bits x_i being either one or zero.

Let us define some more notation. We number the variable nodes with numbers $1, \ldots, N$ (corresponding to the bits in $\mathbf{x}$) and the check nodes with numbers $1, \ldots, L$ (i.e., the number of the row in $\mathbf{H}$). The set $M(i)$ is the set of numbers of all check nodes adjacent to the variable node i, and $N(j)$ is the set of numbers of variable nodes adjacent to the check node j.

3.2.2. The Non-log LDPC Decoder

Each node in the graph has messages it is passing on to its neighbor nodes. Each variable node i passes to its neighbor j a message consisting of a pair of numbers q_{ji}^0, q_{ji}^1. These numbers are essentially probabilities of x_i being either 0 or 1, respectively, given information from all the other check nodes $M(i)\backslash j$ that their respective check constraints are satisfied. Each check node j also passes messages r_{ji}^0, r_{ji}^1 corresponding to probabilities that the jth parity-check constraint is satisfied given $x_i = 0$ or $x_i = 1$, respectively. Each variable node also has overall probabilities q_i^0, q_i^1 associated with it; these probabilities will, when we are done, be our *a poste-*

riori probability outputs of the LDPC decoder. The q_i^0 and q_i^1 are sometimes called **pseudo-posterior** probabilities because of this. Note that

$$\begin{aligned} q_{ji}^0 + q_{ji}^1 &= 1 \\ r_{ji}^0 + r_{ji}^1 &= 1 \\ q_i^0 + q_i^1 &= 1 \end{aligned} \tag{3.8}$$

as one might expect.

Saying that the jth parity-check constraint is satisfied is equivalent to saying

$$\bigoplus_{k \in N(j)} x_k = 0 \tag{3.9}$$

or

$$x_i = \bigoplus_{k \in N(j) \backslash i} x_k \quad . \tag{3.10}$$

Hence, we can define r_{ji}^0, r_{ji}^1 as

$$\begin{aligned} r_{ji}^0 &= P\left(0 = \bigoplus_{k \in N(j) \backslash i} x_k \;\middle|\; F \right) \\ r_{ji}^1 &= P\left(1 = \bigoplus_{k \in N(j) \backslash i} x_k \;\middle|\; F \right) \end{aligned} \tag{3.11}$$

where F is shorthand for the set of initial *a priori* information our LDPC decoder starts with. We compute the r_{ji}^0, r_{ji}^1 values from the messages q_{ji}^0, q_{ji}^1 using the following trick. Consider a set of independent random bits $y_1, \ldots, y_M$ and suppose we wish to find the probability $P(\bigoplus_{i=1}^M y_i = 0)$. Define the partial sums

$$z_n \equiv \bigoplus_{i=1}^{n} y_i \qquad n = 0, 1, \ldots, M \tag{3.12}$$

where we define the empty partial sum $z_0 = 0$. Note that we have a recursion

$$z_n = z_{n-1} \oplus y_n \quad n = 1, \ldots, M \quad . \tag{3.13}$$

Now consider that for two independent random bits a, b with known probabilities $P(a = 1), P(b = 1)$ we have

$$\begin{aligned} P(a \oplus b = 0) &= P(a = 0)P(b = 0) + P(a = 1)P(b = 1) \\ &= (1 - P(a = 1))(1 - P(b = 1)) + P(a = 1)P(b = 1) \\ &= 2P(a = 1)P(b = 1) - P(a = 1) - P(b = 1) + 1 \end{aligned} \tag{3.14}$$

and hence

$$2P(a \oplus b = 0) - 1 = (1 - 2P(a = 1))(1 - 2P(b = 1)) \quad . \tag{3.15}$$

Now z_{n-1} and y_n are by assumption independent, so we can apply the previous equation to get

$$\begin{aligned} 2P(z_n = 0) - 1 &= 2P(z_{n-1} \oplus y_n = 0) - 1 \\ &= (1 - 2P(z_{n-1} = 1))(1 - 2P(y_n = 1)) \\ &= (2P(z_{n-1} = 0) - 1)(1 - 2P(y_n = 1)) \end{aligned} \tag{3.16}$$

and, since $2P(z_0 = 0) - 1 = 2(1) - 1 = 1$, we can recursively compute

$$2P(z_M = 0) - 1 = \prod_{i=1}^{M} (1 - 2P(y_i = 1)) \tag{3.17}$$

and hence

$$P\left(z_M = \left\{ \begin{array}{c} 0 \\ \\ 1 \end{array} \right\}\right) = \frac{1}{2}\left(1 \pm \prod_{i=1}^{M}(1 - 2P(y_i = 1))\right) \quad . \tag{3.18}$$

Applying (3.18) to the problem of computing the r_{ji}^0, r_{ji}^1 while taking q_{ji}^0, q_{ji}^1 as the probabilities associated with bit x_i, we get

$$\begin{aligned} r_{ji}^0 &= \frac{1}{2}\left(1 + \prod_{k \in N(j)\backslash i} \delta q_{jk}\right) \\ r_{ji}^1 &= \frac{1}{2}\left(1 - \prod_{k \in N(j)\backslash i} \delta q_{jk}\right) \end{aligned} \tag{3.19}$$

where we define the quantities δq_{ji} as follows:

$$\delta q_{ji} = 1 - 2q_{ji}^1 = q_{ji}^0 - q_{ji}^1 \quad . \tag{3.20}$$

Note that this presumes that all the variable nodes, all the x_i connected to a given check node are in fact independent. This is not necessarily the case if the graph of our code includes **cycles**, paths that loop back on themselves. Hence, if we have cycles in our graph, the assumptions upon which the belief propagation decoder is based are not valid, and the decoder may not work as well [6]. As we shall see in later chapters, trying to reduce or eliminate cycles from our codes is an important part of designing LDPC codes.

Given the r_{ji}^0, r_{ji}^1 values computed above, we can now compute the pseudo-posterior probabilities for each bit (i.e., each variable node) as follows, where C_i denotes the event that all the parity-checks $N(i)$ that bit i participates in are satisfied, and

$b \in \mathbb{Z}_2$:

$$\begin{aligned} q_i^b &= P(x_i = b|C_i, F) \\ &= \frac{P(C_i|x_i = b, F)P(x_i = b|F)P(F)}{P(C_i, F)} \end{aligned} \tag{3.21}$$

which follows from Bayes' Theorem. Since $q_i^0 + q_i^1 = 1$, we can rewrite (3.21) as

$$q_i^b = \alpha\left(P(C_i|x_i = b, F)P(x_i = b|F)\right) \tag{3.22}$$

where the operator $\alpha()$ normalizes its argument by multiplying it by a constant such that we get a valid probability distribution, i.e., that the values over all b sum to one. We threw away some terms from (3.21), namely the $p(F)$ and $p(C_i, F)$ terms, but since they do not depend on b, they can be thought of as disappearing into the normalization $\alpha()$. Now we have

$$\begin{aligned} P(C_i|x_i = b, F) &= \prod_{j \in M(i)} P\left(0 = \bigoplus_{k \in N(j)} x_k \;\middle|\; x_i = b, F \right) \\ &= \prod_{j \in M(i)} P\left(b = \bigoplus_{k \in N(j) \setminus i} x_k \;\middle|\; F \right) \\ &= \prod_{j \in M(i)} r_{ji}^b \end{aligned} \tag{3.23}$$

and $P(x_i = b|F)$ are just our *a priori* probabilities for each bit. Let us for simplicity define

$$p_i^b \equiv P(x_i = b|F) \quad . \tag{3.24}$$

We now can write the full equation for the pseudo-posterior probabilities by

combining (3.22) and (3.23):

$$q_i^b = \alpha \left(p_i^b \prod_{j \in M(i)} r_{ji}^b \right) \tag{3.25}$$

and, since the q_{ji}^b are defined as containing the information from all the check node neighbors of i **except** for j, we must exclude the contribution from the message r_{ji}^b from node j to node i to get

$$q_{ji}^b = \alpha \left(p_i^b \prod_{k \in M(i) \setminus j} r_{ki}^b \right) \quad . \tag{3.26}$$

We now can describe the belief propagation LDPC decoder algorithm. We start with *a priori* probabilities p_i^b and proceed thus:

1. Initialize all the initial variable node messages from the *a priori* probabilities:

$$q_{ji}^b = p_i^b \quad . \tag{3.27}$$

2. Compute all the r_{ji}^b using (3.19).

3. Compute the q_{ji}^b with (3.26) and the pseudo-posterior probabilities q_i^b with (3.25).

4. From the pseudo-posterior probabilities compute a provisional hard decoding $\hat{\mathbf{x}}$ of the codeword with

$$\hat{x}_i = \begin{cases} 1 & \text{if } q_i^1 > 0.5 \\ 0 & \text{if } q_i^1 \leq 0.5 \end{cases} \quad . \tag{3.28}$$

5. Check to see if $\mathbf{H}\hat{\mathbf{x}} = 0$. If so, we have a valid codeword and we are done. Otherwise, go back to Step 2.

When the algorithm terminates, our final *a posteriori* probabilities are the q_i^b values. In practice, of course, we impose some limit on the total number of iterations to prevent the decoding algorithm from looping indefinitely. If we have done, say, 100 iterations and still have not found a valid codeword, we give up and report an error. One of the nice things about the LDPC decoder is that if it fails to find a valid codeword, it can report that the word is not valid; compare this to the decoder for turbo codes [14] which can produce words that are not valid codewords without any sort of error indication.

3.2.3. The Log LDPC Decoder

We now derive the log domain version of the LDPC decoder. First let us, in accordance with Section 2.1, define the log domain version of r_{ji}^b,

$$L(r_{ji}) = \log \frac{r_{ji}^0}{r_{ji}^1} \tag{3.29}$$

and define $L(q_{ji})$ and $L(q_i)$ in analogous fashion. From (3.19), we find that

$$L(r_{ji}) = \log \frac{1 + \prod_{k \in N(j)\backslash i} \delta q_{ji}}{1 - \prod_{k \in N(j)\backslash i} \delta q_{ji}} \quad . \tag{3.30}$$

Now, since

$$\begin{aligned}
\delta q_{ji} &= q_{ji}^0 - q_{ji}^1 \\
&= \frac{\exp(L(q_{ji})/2)}{\exp(L(q_{ji})/2) + \exp(-L(q_{ji})/2)} - \frac{\exp(-L(q_{ji})/2)}{\exp(L(q_{ji})/2) + \exp(-L(q_{ji})/2)} \\
&= \frac{\exp(L(q_{ji})/2) - \exp(-L(q_{ji})/2)}{\exp(L(q_{ji})/2) + \exp(-L(q_{ji})/2)} \\
&= \frac{\exp(L(q_{ji})) - 1}{\exp(L(q_{ji})) + 1} \\
&= \tanh\left(\frac{L(q_{ji})}{2}\right)
\end{aligned} \tag{3.31}$$

we can rewrite (3.30) as

$$\begin{aligned} L(r_{ji}) &= \log \frac{1+\prod_{k\in N(j)\backslash i} \tanh(L(q_{jk})/2)}{1-\prod_{k\in N(j)\backslash i} \tanh(L(q_{jk})/2)} \\ &= 2\tanh^{-1}\left(\prod_{k\in N(j)\backslash i} \tanh(L(q_{jk})/2)\right) \end{aligned} \tag{3.32}$$

which gives us the log form of the rule for updating the messages from the check nodes. We now need log forms of (3.25) and (3.26); a few moment's computation shows that

$$L(q_{ji}) = L(p_i) + \sum_{k\in M(i)\backslash j} L(r_{ki}) \tag{3.33}$$

and

$$L(q_i) = L(p_i) + \sum_{j\in M(i)} L(r_{ji}) \quad . \tag{3.34}$$

The log version of the algorithm proceeds similarly to the non-log version:

1. Initialize all the initial variable node messages from the *a priori* likelihoods:

$$L(q_{ji}) = L(p_i) \quad . \tag{3.35}$$

2. Compute all the $L(r_{ji})$ using (3.32).

3. Compute the $L(q_{ji})$ with (3.33) and the pseudo-posterior likelihoods $L(q_i)$ with (3.34).

4. From the pseudo-posterior likelihoods compute a provisional hard decoding $\hat{\mathbf{x}}$ of the codeword with

$$\hat{x}_i = \begin{cases} 1 & \text{if } L(q_i) < 0 \\ 0 & \text{if } L(q_i) > 0 \end{cases} \quad . \tag{3.36}$$

5. Check to see if $\mathbf{H}\hat{\mathbf{x}} = 0$. If so, we have a valid codeword and we are done. Otherwise, go back to Step 2.

3.3. Optimizations and Approximations

The equation (3.32) is somewhat computationally expensive, requiring several multiplications and hyperbolic tangents for each $L(r_{ji})$ evaluated. We can rearrange this equation to speed things up a bit. First note that any product of terms a_i can be rearranged in terms of an exponential of sums of logs like this

$$\prod_i a_i = \prod_i \text{sgn}(a_i) \exp(\sum_i \log(|a_i|)) \quad . \tag{3.37}$$

Since tanh and $\tanh^{-1}$ are both odd functions, we can freely move terms that are ± 1 in value past these functions:

$$\tanh(\pm x) = \pm \tanh(x) \qquad \text{and} \qquad \tanh^{-1}(\pm x) = \pm \tanh^{-1}(x) \tag{3.38}$$

Applying both (3.37) and (3.38) to (3.32) gives us

$$\begin{aligned} L(r_{ji}) &= 2\tanh^{-1}\left(\prod_{k\in N(j)\backslash i} \tanh(L(q_{ji})/2)\right) \\ &= \prod_{k\in N(j)\backslash i} \text{sgn}(\tanh(L(q_{ji})/2)) \\ &\quad \times 2\tanh^{-1}\left(\exp\left(\sum_{k\in N(j)\backslash i} \log(\tanh(|L(q_{ji})/2|))\right)\right) \end{aligned} \tag{3.39}$$

which simplifies to

$$L(r_{ji}) = \prod_{k \in N(j) \backslash i} \operatorname{sgn}(L(q_{ji})) \times 2 \tanh^{-1} \left(\exp \left(\sum_{k \in N(j) \backslash i} \log(\tanh(|L(q_{ji})/2|)) \right) \right). \quad (3.40)$$

Now let us define a function $\Psi(x) = -\log(\tanh(x/2))$. Then we have

$$\Psi^{-1}(y) = 2 \tanh^{-1} \exp(-y) \quad (3.41)$$

and we can rewrite (3.40) as

$$\begin{aligned} L(r_{ji}) &= \prod_{k \in N(j) \backslash i} \operatorname{sgn}(L(q_{ji})) \times 2 \tanh^{-1} \left(\exp \left(\sum_{k \in N(j) \backslash i} -\Psi(|L(q_{ji})/2|) \right) \right) \\ &= \prod_{k \in N(j) \backslash i} \operatorname{sgn}(L(q_{ji})) \Psi^{-1} \left(\sum_{k \in N(j) \backslash i} \Psi(|L(q_{ji})/2|) \right) . \end{aligned} \quad (3.42)$$

We have thus replaced $|N(j) \backslash i| - 1$ multiplications with the same number of additions (plus some multiplications of the $\operatorname{sgn}(L(q_{ji}))$ terms, but those are trivial since the numbers are always ± 1). Also, as Gallager pointed out [5], Ψ is its own inverse, i.e.,

$$\Psi^{-1}(x) = \Psi(x) . \quad (3.43)$$

Hence we only need be able to do additions and this one special function $\Psi(x)$ to compute the $L(r_{ji})$, and we can, if more speed is needed, use a lookup table to compute $\Psi(x)$.

For even further increases in speed, we can resort to approximation techniques similar to those discussed in the previous chapter for the BCJR algorithm. This approximate version of the LDPC decoder is sometimes called the **max-product**

algorithm. The basic idea behind this approximation is this: consider the computation of the r_{ji}^b associated with some check node j. Suppose check node j has degree C, and let the indices of the bits participating in the check j be $a_1, a_2, \ldots, a_C$. Without loss of generality we can assume $a_1 = i$. Then we can rephrase the definition of the r_{ji}^b (3.11) as

$$r_{ji}^b = P\left(b = \bigoplus_{k=2}^{C} x_{a_k} \;\middle|\; F \right) \tag{3.44}$$

and note that this probability is the sum of the probabilities for each possible combination of bits x_{a_k} that satisfy the overall parity constraint. That is to say, if we define the set of combinations of bits

$$D_b = \left\{ (x_{a_2}, \ldots, x_{a_C}) | b = \bigoplus_{k=2}^{C} x_{a_k} \right\} \tag{3.45}$$

then

$$r_{ji}^b = \sum_{(x_{a_2},\ldots,x_{a_C}) \in D_b} P(x_{a_2}, \ldots, x_{a_C} | F) \tag{3.46}$$

which in turn can be rewritten

$$r_{ji}^b = \sum_{(x_{a_2},\ldots,x_{a_C}) \in D_b} \prod_{k=2}^{C} q_{jk}^{x_{a_k}} \quad . \tag{3.47}$$

The critical idea behind the max-product approximation is to assume that one of the terms of the above sum is dominant over the others and to approximate r_{ij}^b as the largest of these terms, so instead of (3.47) we compute

$$r_{ji}^b = \alpha \left(\max_{(x_{a_2},\ldots,x_{a_C}) \in D_b} \prod_{k=2}^{C} q_{jk}^{x_{a_k}} \right) \tag{3.48}$$

where we have added an $\alpha()$ normalization to compensate for the fact that our approximation might give probabilities that do not sum exactly to one.

The above expression does not look at first to be simpler, for though it uses

a max operation instead of a sum, it requires us to compute probabilities for all the possible bit combinations in each set D_b, something which we avoided in (3.19) through a clever trick. However, it turns out that in the log-domain version of the max-product algorithm, we can in fact derive a really simple version of the update rule for the $L(r_{ji})$ likelihoods. To see this, consider the case where check node j has degree three, and in order to simplify the notation a bit let us assume the indices of the bits in this node are $a_k = 0, 1, 2$. Then our above equation gives us

$$\begin{aligned} r_{j0}^0 &= \alpha \max(q_{j1}^0 q_{j2}^0, q_{j1}^1 q_{j2}^1) \\ r_{j1}^1 &= \alpha \max(q_{j1}^0 q_{j2}^1, q_{j1}^1 q_{j2}^0) \end{aligned} \tag{3.49}$$

for some normalizing constant α. We now must figure out how to write this in terms of log-likelihood ratios. For convenience, let us call $L(q_{j1}) = a$ and $L(q_{j2}) = b$, and let us define

$$\begin{aligned} a_0 &= q_{j1}^0 \\ a_1 &= q_{j1}^1 \\ b_0 &= q_{j2}^0 \\ b_1 &= q_{j2}^1 \quad . \end{aligned} \tag{3.50}$$

Note that

$$\frac{a_0 b_0}{a_1 b_1} = \exp(a)\exp(b) = \exp(a+b) \tag{3.51}$$

so we can in general write

$$r_{j0}^0 = \begin{cases} \alpha a_0 b_0 & \text{if } a+b > 0 \\ \alpha a_1 b_1 & \text{if } a+b < 0 \\ \alpha a_0 b_0 = \alpha a_1 b_1 & \text{if } a+b = 0 \end{cases} \tag{3.52}$$

and similarly

$$\frac{a_0 b_1}{a_1 b_0} = \frac{\exp(a)}{\exp(b)} \tag{3.53}$$

leads to

$$r_{j0}^1 = \begin{cases} \alpha a_0 b_1 & \text{if } a - b > 0 \\ \alpha a_1 b_0 & \text{if } a - b < 0 \\ \alpha a_0 b_1 = \alpha a_1 b_0 & \text{if } a - b = 0 \end{cases} \tag{3.54}$$

no matter what a, b are. We shall use these results later. We now have several cases:

Case 1: $|a| > |b|, a > 0$. We have $a + b > 0$ and $a - b > 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_0 b_0 \\ r_{j0}^1 &= \alpha a_0 b_1 \\ L(r_{j0}) &= b \quad . \end{aligned} \tag{3.55}$$

Case 2: $|a| > |b|, a < 0$. We have $a + b < 0$ and $a - b < 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_1 b_1 \\ r_{j0}^1 &= \alpha a_1 b_0 \\ L(r_{j0}) &= -b \quad . \end{aligned} \tag{3.56}$$

Case 3: $|a| < |b|, b > 0$. We have $a + b > 0$ and $a - b < 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_0 b_0 \\ r_{j0}^1 &= \alpha a_1 b_0 \\ L(r_{j0}) &= a \quad . \end{aligned} \tag{3.57}$$

Case 4: $|a| < |b|, b < 0$. We have $a + b < 0$ and $a - b > 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_1 b_1 \\ r_{j0}^1 &= \alpha a_0 b_1 \\ L(r_{j0}) &= -a \end{aligned} \tag{3.58}$$

Case 5: $|a| = |b| = 0$. We have $a = b = 0$, so $a_1 = b_1 = a_0 = b_0 = 1/2$, so

$$\begin{aligned} r_{j0}^0 &= \alpha \frac{1}{4} \\ r_{j0}^1 &= \alpha \frac{1}{4} \\ L(r_{j0}) &= 0 \end{aligned} \tag{3.59}$$

Case 6: $|a| = |b|, a > 0, b > 0$. We have $a + b > 0$ and $a - b = 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_0 b_0 \\ r_{j0}^1 &= \alpha a_1 b_0 \\ L(r_{j0}) &= a \end{aligned} \tag{3.60}$$

Case 7: $|a| = |b|, a < 0, b > 0$. We have $a + b = 0$ and $a - b < 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_0 b_0 \\ r_{j0}^1 &= \alpha a_1 b_0 \\ L(r_{j0}) &= a \end{aligned} \tag{3.61}$$

Case 8: $|a| = |b|, a > 0, b < 0$. We have $a + b = 0$ and $a - b > 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_1 b_1 \\ r_{j0}^1 &= \alpha a_0 b_1 \\ L(r_{j0}) &= -a \end{aligned} \tag{3.62}$$

Case 9: $|a| = |b|, a < 0, b < 0$. We have $a + b < 0$ and $a - b = 0$, so

$$\begin{aligned} r_{j0}^0 &= \alpha a_1 b_1 \\ r_{j0}^1 &= \alpha a_0 b_1 \\ L(r_{j0}) &= -a \end{aligned} \tag{3.63}$$

After considering all nine cases, we find that we can define a function, which we call the $\mathcal{M}$-function,

$$\mathcal{M}(a, b) = \operatorname{sgn}(a) \operatorname{sgn}(b) \min(|a|, |b|) \tag{3.64}$$

such that

$$L(r_{j0}) = \mathcal{M}(a, b) \qquad \forall a, b \in \mathbb{R} \quad . \tag{3.65}$$

The $\mathcal{M}$-function can be considered an approximation to the so-called $\mathcal{R}$-function

$$\mathcal{R}(a, b) = 2 \tanh^{-1}(\tanh(a/2) \tanh(b/2)) \quad . \tag{3.66}$$

This approximation gets better for larger $|a|$ or $|b|$. Note that for this case of the degree three check node the corresponding equation from the sum-product algorithm (3.32) gives us $L(r_{j0}) = \mathcal{R}(a, b)$. (This notation of the $\mathcal{M}$-function and $\mathcal{R}$-function is taken from [15].) For the sum-product algorithm, we can consider the $\mathcal{R}$-function

as a dyadic operator, like addition, and rewrite (3.32) as

$$L(r_{ji}) = \underset{k \in N(j) \setminus i}{\mathcal{R}} L(q_{ji}) \tag{3.67}$$

(note that

$$\mathcal{R}(a, \mathcal{R}(b, c)) = \mathcal{R}(\mathcal{R}(a, b), c) \qquad \forall a, b, c \in \mathbb{R} \tag{3.68}$$

so $\mathcal{R}$ is indeed an associative operator). The corresponding equation for the max-product algorithm for any check node j is just

$$L(r_{ji}) = \underset{k \in N(j) \setminus i}{\mathcal{M}} L(q_{ji}) \tag{3.69}$$

so that the max-product algorithm in the log domain is just the sum-product algorithm with this peculiar $\mathcal{R}$ operation replaced by the $\mathcal{M}$ operation, which is easily computed.

3.4. How the LDPC and BCJR Decoders Work Together: Extrinsic Information

The preceding discussion in Chapter 1 implied that the LDPC and BCJR decoders feed each other the *a posteriori* probabilities they produce or, in the log domain, the *a posteriori* likelihood values. This is in fact a bit of an over-simplification. In practice, the values that are passed are not the *a posteriori* likelihoods, but a somewhat modified version of them called the **extrinsic information**. Suppose the LDPC decoder takes as inputs likelihood values $L_{\text{LDPCprior}}(i)$ for each bit i and computes *a posteriori* likelihoods $L_{\text{LDPCpost}}(i)$ using either the sum-product or max-product algorithms as explained above. We then feed into the next iteration of the

BCJR decoder not $L_{\text{LDPCpost}}(i)$, but the extrinsic information

$$L_{\text{LDPCext}}(i) = L_{\text{LDPCpost}}(i) - L_{\text{LDPCprior}}(i) \quad . \tag{3.70}$$

This extrinsic information becomes the input $L_{\text{BCJRprior}}(i)$ to the next round of the BCJR decoder, and the resulting *a posteriori* likelihoods from the BCJR decoder have the prior likelihoods subtracted to yield extrinsic information

$$L_{\text{BCJRext}}(i) = L_{\text{BCJRpost}}(i) - L_{\text{BCJRprior}}(i) \tag{3.71}$$

which becomes input for the next LDPC round, and so on. Once all rounds of turbo equalization are completed, we take the final sets of extrinsic information from each decoder and sum them to get an overall likelihood

$$L(i) = L_{\text{BCJRext}}(i) + L_{\text{LDPCext}}(i) + L_{\text{initialprior}}(i) \tag{3.72}$$

where $L_{\text{initialprior}}(i)$ is the set of initial prior likelihoods we started the first BCJR iteration with (usually all-zeros, since we assume our bits are equally likely). We then hard-decode $L(i)$ to produce final bit decisions.

Why is this subtraction process done to generate the extrinsic information? There are heuristic arguments that each decoder should not be given as input information that originated with a previous iteration of itself, so that each decoder only gets what new information was added by the other decoder, i.e., the extrinsic information from the other decoder. We ourselves do not find these arguments all that convincing. Fan [16] gives a more in-depth argument proceeding as follows. Consider the basic Bayesian equation for the probability that bit x_i has value b

given that the codeword satisfies a constraint C and other prior information F:

$$P(x_i = b|C,F) = \frac{P(C|x_i = b,F)P(x_i = b|F)P(F)}{P(C,F)} \tag{3.73}$$

(note that (3.21) is a special case of this). Putting this in the log-likelihood domain, and noting that the terms that do not depend on b drop out, we get

$$\overbrace{L(x_i|C,F)}^{\text{posterior}} = \overbrace{L(C|x_i,F)}^{\text{extrinsic}} + \overbrace{L(x_i|F)}^{\text{prior}} \tag{3.74}$$

where the various conditional likelihood ratios are defined, in an extension of our previous notation, as follows:

$$\begin{aligned} L(x_i|C,F) &= \log\frac{P(x_i = 0|C,F)}{P(x_i = 1|C,F)} \\ L(C|x_i = b,F) &= \log\frac{P(C|x_i = 0,F)}{P(C|x_i = 1,F)} \\ L(x_i|F) &= \log\frac{P(x_i = 0|F)}{P(x_i = 1|F)} \end{aligned} \quad . \tag{3.75}$$

Here $L(x_i|C,F)$ is the log-likelihood version of the *a posteriori* probability of bit x_i being zero or one given the constraint C, $L(x_i|F)$ is our *a priori* log-likelihood, and the difference, the so-called extrinsic information, is just the log version of the flipped-around conditional probabilities $P(C|x_i = b,F)$. It is thus more reasonable that in order to combine several (independent, we assume) constraints from several different decoders we take the product of the conditional probabilities $P(C_{\text{LDPC}}|x_i = b,F)$, $P(C_{\text{BCJR}}|x_i = b,F)$ together with the *a priori* probabilities $P(x_i = b|F)$ to get final *a posteriori* values, or in the log domain, we get (3.72). Fan [16] notes that in fact we carry out this extrinsic subtraction and combining of several extrinsic values from different constraints in the internal workings of the LDPC decoder as well as between the LDPC decoder and other decoders; consider, e.g., (3.33) and

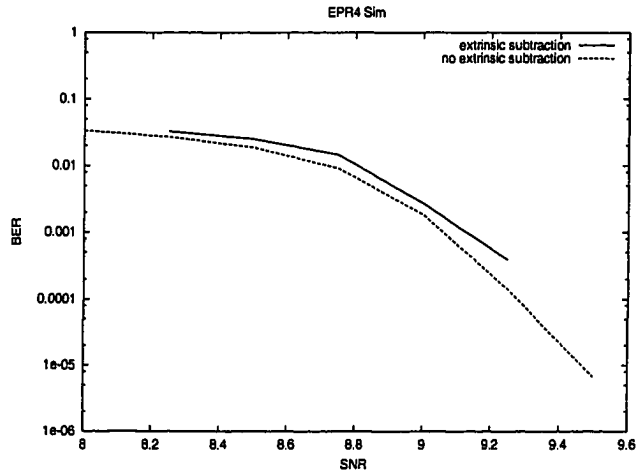


Figure 3.2: Performance of LDPC/BCJR decoder with and without extrinsic subtraction.

(3.34).

The process of subtracting to get the extrinsic information was first done in the realm of turbo code decoding [14], where decoding is done by means of two BCJR decoders iterating back and forth. The above argument due to Fan seems to be reasonable justification for doing this with the LDPC/BCJR decoder combination as well. However, there is a problem. Before we had come across Fan's work [16] we decided to try some simulations to see if the use of this extrinsic subtraction empirically performs better than the simple approach of just passing the *a posteriori* outputs $L_{\text{BCJRpost}}(i), L_{\text{LDPCpost}}(i)$ as the inputs to the other decoders and taking the final *a posteriori* value as our final result. The results of those simulations are shown in Fig. 3.2. We simulated performance of the LDPC code used in [9] over the EPR4 channel at various signal-to-noise ratios, with and without the extrinsic subtraction being used. Curiously, the non-extrinsic-subtraction case performs **better** than the extrinsic-subtraction case. The difference is only about 0.05dB or so, but it does perform better. Clearly, the whole issue of extrinsic information is not understood as well as it could or should be.

Chapter 4

Creating LDPC Codes with Desired Weight Distributions

In this chapter, we explain how to create a random LDPC code matrix $\mathbf{H}$ given the desired code size parameters N (codeword length) and K (user data length), as well as specifying the column and row weights of the $\mathbf{H}$ matrix. The N and K values are pretty much fixed by the requirements of our magnetic recording system (see Chapter 1), but the choice of column and row weight distributions is up to the user. Figuring out which column and row weight distributions are best for a given application is an important problem (in fact, it is **the** main problem of this dissertation), but it is a problem we will defer to later chapters for the moment. For now, we assume that we have been given column and row weight distributions, and consider the problem of how to make an LDPC code fitting those constraints.

4.1. Weight Distributions

The column and row weight distributions are conventionally [15] specified by vectors of numbers λ_i and ρ_i. λ_i is the fraction of one bits in the $\mathbf{H}$ matrix of the code that occur in columns of weight i, and similarly ρ_i is the fraction of one bits that occur in rows of weight i. That is to say, if $\lambda_3 = 1/4$, then one-quarter of all the one bits in the $\mathbf{H}$ matrix are found in columns of total weight three. Here i is an integer that is two or larger, since weight zero or one columns or rows are not something we would ever want in the parity-check matrix in practice. Sometimes, instead of giving explicit vectors λ_i, ρ_i, we specify the weight distributions with two polynomials $\lambda(x), \rho(x)$ defined as

$$\begin{aligned} \lambda(x) &= \sum_{i=2}^{\infty} \lambda_i x^{i-1} \\ \rho(x) &= \sum_{i=2}^{\infty} \rho_i x^{i-1} \end{aligned} \tag{4.1}$$

Note that if, say, $\lambda_3 = 1/4$, this is not the same thing as having one-quarter of

all columns have weight three. For example, if $\lambda_3 = 3/7$ and $\lambda_4 = 4/7$ and all other $\lambda_i = 0$, then if the total number of one bits in $\mathbf{H}$ is Q, then there are $3Q/7$ bits in columns of weight three and $4Q/7$ bits in columns of weight four. Hence, the total number of columns of weight three is $(3Q/7)/3 = Q/7$ and for weight four we have $(4Q/7)/4 = Q/7$ columns, so the fraction of columns with weight three is 1/2, and the same is true for weight four. In general given a column weight distribution λ_i , the fraction of columns that have weight i is

$$\hat{\lambda}_i = \frac{\lambda_i/i}{\sum_{i=2}^{\infty} \lambda_i/i} \tag{4.2}$$

and a similar equation holds for the fraction of rows with a given weight:

$$\hat{\rho}_i = \frac{\rho_i/i}{\sum_{i=2}^{\infty} \rho_i/i} \quad . \tag{4.3}$$

Why do we specify the weight distributions in this somewhat counter-intuitive way, as λ_i values rather than $\hat{\lambda}_i$ values? As we shall see in later chapters, the λ_i and ρ_i form of the weight distribution is more convenient for analyzing the performance of LDPC codes through density evolution [17]. For actually creating an LDPC code meeting these parameters, the $\hat{\lambda}_i, \hat{\rho}_i$ version of these parameters are more useful; we can freely convert between the two using (4.2) and (4.3).

Note that there are constraints on the λ_i and ρ_i; we are not entirely free to choose any values we want. The definitions of the λ_i and ρ_i give these obvious constraints:

$$\sum_{i=2}^{\infty} \lambda_i = \sum_{i=2}^{\infty} \rho_i = 1 \tag{4.4}$$

and

$$\lambda_i, \rho_i \in [0, 1] \qquad \forall i \quad . \tag{4.5}$$

What may not be so obvious is that the code rate $R = K/N$ also constrains our

λ_i, ρ_i values. The reason is this: $\mathbf{H}$ has Q one bits in it, arranged in N columns and $L = N - K$ rows. We know that the Q bits must be in columns of various weights:

$$\begin{aligned} Q &= N\sum_{i=2}^{\infty} i\hat{\lambda}_i \\ &= N\frac{\sum_{i=2}^{\infty}\lambda_i}{\sum_{i=2}^{\infty}\lambda_i/i} \\ &= \frac{N}{\sum_{i=2}^{\infty}\lambda_i/i} \end{aligned} \tag{4.6}$$

but these same Q bits are in the rows of various weights as well:

$$\begin{aligned} Q &= L\sum_{i=2}^{\infty} i\hat{\rho}_i \\ &= L\frac{\sum_{i=2}^{\infty}\rho_i}{\sum_{i=2}^{\infty}\rho_i/i} \\ &= \frac{L}{\sum_{i=2}^{\infty}\rho_i/i} \end{aligned} \tag{4.7}$$

and hence

$$\frac{N}{\sum_{i=2}^{\infty}\lambda_i/i} = \frac{L}{\sum_{i=2}^{\infty}\rho_i/i} \quad . \tag{4.8}$$

But $L = (1 - R)N$, so we have

$$\frac{1}{\sum_{i=2}^{\infty}\lambda_i/i} = \frac{1-R}{\sum_{i=2}^{\infty}\rho_i/i} \tag{4.9}$$

and hence

$$R = 1 - \frac{\sum_{i=2}^{\infty}\rho_i/i}{\sum_{i=2}^{\infty}\lambda_i/i} = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx} \quad . \tag{4.10}$$

4.2. Specifying LDPC Codes as Permutations

As we saw in the previous chapter, each parity-check matrix $\mathbf{H}$ corresponds to a network of variable and check nodes, as in Fig. 3.1. Each link in that network links

a check node with a variable node; each bit in $\mathbf{H}$ links a row with a column. Let r_i be the weight of the ith row of $\mathbf{H}$, and similarly define c_i on the columns. Let us define two vectors

$$\mathbf{R} = \left[\overbrace{1,\ldots,1}^{r_1},\overbrace{2,\ldots,2}^{r_2},\ldots,\overbrace{L,\ldots,L}^{r_L}\right] \tag{4.11}$$

and

$$\mathbf{C} = \left[\overbrace{1,\ldots,1}^{c_1},\overbrace{2,\ldots,2}^{c_2},\ldots,\overbrace{N,\ldots,N}^{c_N}\right] \tag{4.12}$$

where each number i occurs r_i times in $\mathbf{R}$ and similarly for $\mathbf{C}$. Note that $\mathbf{R}$ and $\mathbf{C}$ both have total number of elements Q. Then $\mathbf{H}$ can be thought of as specifying a bijection between members of $\mathbf{R}$ and members of $\mathbf{C}$. Hence, each matrix $\mathbf{H}$ can be specified by some permutation $\pi \in \mathrm{Sym}(Q)$, the set of permutations on Q symbols. For some numbering of the Q bits in $\mathbf{H}$, we have that the ith bit of $\mathbf{H}$ links column number C_i and row $R_{\pi(i)}$. Hence we can reduce the problem of generating $\mathbf{H}$ to that of generating a suitable permutation π.

Note that not every permutation $\pi \in \mathrm{Sym}(Q)$ corresponds to a possible $\mathbf{H}$ matrix. Some permutations π might try to have more than one link between the same variable node/check node pair. Mathematically, that would correspond to there being i, j such that $C_i = C_j$ and $R_{\pi(i)} = R_{\pi(j)}$. We need to check any permutation that we generate to make sure that it is **admissible**, i.e., that

$$\text{if} \quad C_i = C_j \quad \text{then} \quad R_{\pi(i)} \neq R_{\pi(j)} \quad \forall i, j = 1,\ldots,Q \quad . \tag{4.13}$$

The simplest way to check this is to initialize a matrix $\mathbf{H}$ to all zeros and for each $i = 1,\ldots,Q$ try to set a bit in the ith column and $\pi(i)$th row; if we end up trying to set a bit that is already set, we have an inadmissible π.

4.3. Cycle Elimination

Even once we have an admissible π, it may lead to a matrix **H** that is not desirable for LDPC decoding. As mentioned in the previous chapter and as discussed in [6], if the network for our **H** matrix has cycles in it, the basic independence assumptions upon which the LDPC belief-propagation decoder was derived are violated. Hence, cycles impair the performance of the decoder. To avoid this impairment, we attempt to remove any such cycles from our code. In practice, it is not practical to remove all cycles. However, the shortest cycles, the ones of length four, are the ones which impair performance the most, so if we can eliminate them, we will have a better code matrix **H**. To eliminate the four-cycles, we first have to find them in our code. Let us define a couple of functions. Rows(S) is a function that maps a column number i to a set of row numbers; this function is defined as

$$\text{Rows}(i) = \left\{r | r = R_{\pi(k)} \text{ for some } k \text{ such that } C_k = i\right\} \tag{4.14}$$

and similarly

$$\text{Cols}(i) = \left\{c | c = C_{\pi^{-1}(k)} \text{ for some } k \text{ such that } R_k = i\right\} \tag{4.15}$$

maps a row number i to a set of column numbers. We can now check π for four-cycles with the following algorithm.

1. Let $i = 1$.
2. Find $j = R_{\pi(i)}$; this is the row on one side of link number i.
3. Compute the set $S_2(j) = \text{Cols}(j) \backslash C_i$.
4. For each $k \in S_2(j)$, compute a set $S_3(k) = \text{Rows}(k) \backslash j$.
5. For each $\ell \in S_3(k)$, compute a set $S_4(\ell) = \text{Cols}(\ell) \backslash k$.

6. If any of the sets $S_4(\ell)$ contain C_i, then we have a cycle $C_i \rightarrow j \rightarrow k \rightarrow \ell \rightarrow C_i$. Stop.

7. Otherwise, let $i \leftarrow i + 1$, and if $i <= Q$, go back to Step 1.

8. We are done! If we got here, there are no four-cycles in the graph.

If we found a four-cycle $C_i \rightarrow j \rightarrow k \rightarrow \ell \rightarrow C_i$, we can attempt to eliminate it by altering the permutation π. We do this by picking a random number $q \in \{1, \ldots, Q\} \backslash i$ and "switching" the links $i \rightarrow j$ and $q \rightarrow \pi(q)$. This corresponds to making a new permutation π' such that

$$\pi'(x) = \begin{cases} \pi(q) & \text{if} \quad x = i \\ j & \text{if} \quad x = q \\ \pi(x) & \text{otherwise} \end{cases} \tag{4.16}$$

Once we have the new permutation, we can go back and check it for four-cycles, and continue this process until we have a π free of four-cycles. Note that for some sets of code parameters N, R, λ_i, ρ_i, we may in fact never find such a cycle-free π; the preceding algorithm may never converge. High code rates (say, $R > 0.9$) and high average column weights ($\sum i\hat{\lambda}_i > 4$) tend to be particularly bad cases for this. In such cases where the algorithm fails to converge, the code designer has little choice but to accept that the generated code is going to have four-cycles. In some cases we have been able to do some limited removal of four-cycles by restricting the preceding algorithm to only a subset of the possible bits i, only trying to remove cycles which go through columns of low weight. What we did in that case was restrict the above cycle search to only those initial i values for which column C_i had weight less than a certain threshold. It is not clear whether this limited cycle removal is of much value with regard to the performance of the resulting code, however.

4.4. Making a Systematic Code

Once we have a permutation π which has been cleaned of four-cycles, we still need to be sure we can make a parity-check matrix $\mathbf{H}$ corresponding to a systematic code. This means, as we saw in the last chapter, that we need an $\mathbf{H}$ whose rightmost $L \times L$ segment is invertible. We first test the rightmost segment of $\mathbf{H}$ for invertibility. If that matrix is invertible, we are done. Otherwise, we have to rearrange columns in $\mathbf{H}$ until we have a suitably invertible RHS; doing Gaussian elimination on the entire matrix $\mathbf{H}$ and paying attention to the column reordering moves can tell us which columns need to be shuffled into the RHS to make it invertible. Gaussian elimination can also tell us if we are in the unfortunate situation where $\mathbf{H}$ is not full rank, and **no** rearrangement of columns can ever give us an invertible RHS. This is pretty rare for irregularly distributed $\mathbf{H}$ (i.e. where λ_i is not a delta function). However, for an important case of regular codes, codes of uniform column weight four ($\lambda_i = \delta_{i4}$), the RHS can never be invertible. This is because, if we have a square matrix whose columns all have even parity, if we try to invert it by doing Gaussian elimination on the columns, adding columns to each other will never give columns with odd parity, so we can never get the identity matrix. In such cases where we do not have an $\mathbf{H}$ matrix of full rank, all one can do is randomly add an extra bit somewhere to $\mathbf{H}$ to make it full rank. Note that this does make $\mathbf{H}$ violate the given weight constraints λ_i, ρ_i slightly, and may also spoil the four-cycle freedom.

4.5. Summary

We are now in a position to list the overall algorithm for generating random parity-check matrices $\mathbf{H}$. We proceed as follows:

1. Convert our given λ_i, ρ_i into ratios of how many columns/rows have weight i, $\hat{\lambda}_i, \hat{\rho}_i$, as per (4.2) and (4.3).

2. Create a random set of row weights r_i and column weights c_i that match the $\hat{\rho}_i$ and $\hat{\lambda}_i$ distributions. Also create the initial vectors $\mathbf{R}$, $\mathbf{C}$ as in (4.11), (4.12). Also compute $Q = \sum_i r_i = \sum_i c_i$.

3. Create a random permutation $\pi \in \mathrm{Sym}(Q)$.

4. Check π for admissibility as in Section 4.2. If π is not admissible, randomly pick two links in π to exchange as in (4.16) and go back to Step 4.

5. Check π for four-cycles as in Section 4.3. If we find a cycle through link i, randomly exchange that link with some other link q as in (4.16), and go back to Step 4.

6. Create the matrix $\mathbf{H}$ from π, and check its RHS for invertibility. If it is invertible, we are done.

7. Otherwise, do Gaussian elimination on $\mathbf{H}$. If it succeeds ($\mathbf{H}$ is full rank), reorder columns as needed to make the RHS of $\mathbf{H}$ invertible, and we are done.

8. Otherwise, we are in a bit of trouble. $\mathbf{H}$ is not full rank. Add a random bit to $\mathbf{H}$, compute the corresponding permutation π, adjust the row/column weight tables r_i, c_i accordingly, and go back to Step 4.

This is how we create random LDPC codes meeting specified weight distributions.

Chapter 5

Analyzing Performance of LDPC Codes with Density Evolution

In order to be able to design good LDPC codes and find the λ_i and ρ_i distributions that lead to the best codes, we need some way of estimating how well an LDPC code will perform given only its λ_i, ρ_i values. Fortunately, there is a technique for doing such predictions; this technique, called **density evolution** was developed by Richardson and Urbanke [17]. By studying the behavior of probability density functions (pdfs) of various variables in the LDPC algorithm, one can predict the bit error rate of an LDPC code when used with various memoryless channels. This lets us compute so-called **threshold values** for each set of LDPC code parameters λ_i, ρ_i; loosely speaking, the threshold value tells us how noisy our memoryless channel has to be in order to have significantly high bit error rates. Obviously, one wants to select λ_i, ρ_i sets which give as high a threshold value as one can get. In this chapter, we explain in detail how to do density evolution and find the threshold for a given set of LDPC code parameters. The notation used here loosely follows that in [15].

5.1. Preliminaries: Computing with Probability Density Functions

To do density evolution, we must be able to perform various operations on probability density functions. Ideally, the pdf of some random variable $z \in \mathbb{R}$ is some function $p_z(z)$ where $p_z \in L^1(\mathbb{R})$ and $p_z(z) \in [0,1]$ for all z. However, we would need to have an (uncountably) infinite amount of storage to completely represent a general function p_z which can take on values at any point $z \in \mathbb{R}$. In order to do practical computations with pdfs on our computer, we need to introduce a **quantized** form of the pdf. We pick a quantization interval δ and break up the real axis into intervals of size δ. Our quantized pdf $\mathcal{P}[z]$ is thus a vector with subscript $k \in \mathbb{Z}$

such that

$$\mathcal{P}[z]_k = P\left[k\delta - \frac{\delta}{2} \leq z < k\delta + \frac{\delta}{2}\right] = \int_{k\delta-\frac{\delta}{2}}^{k\delta+\frac{\delta}{2}} p_z(z)dz \quad . \tag{5.1}$$

Obviously, this loses a good bit of information from the original pdf, but like most quantization procedures, one hopes that if δ is chosen sufficiently small, the information loss is acceptable. Note that our quantized pdf $\mathcal{P}[z]$ still would require an infinite amount of storage unless we imposed another condition, namely that $\mathcal{P}[z]_k$ is negligible outside some interval $k \in [K_{\text{low}}, K_{\text{high}}]$. This is easily the case for random variables that represent log-likelihood values; if our (non-zero, finite) likelihood ratio p/q is represented as an IEEE format double-precision number, it is simple to show that the log of that ratio must always satisfy

$$|\log(p/q)| < 708.3 \tag{5.2}$$

and we can always represent the edge cases $p/q = 0, p/q = \infty$ by setting $\log(p/q) = \mp 708.3$, respectively. For other sorts of random variables, such as, e.g., z Gaussian, we can pick $K_{\text{low}}, K_{\text{high}}$ sufficiently far out to include, say, ten sigmas worth of deviation about the mean and consider as negligible the pdf values outside that range. (Note: in [15], the author implicitly assumed throughout that all pdfs were constrained to the same range, with $K_{\text{low}} = -K/2, K_{\text{high}} = K/2 - 1$ for some even integer K. We find it convenient, and not much more complex, to allow the possibility that K_{low} and K_{high} may be different for different pdfs.)

Note that since we assume our random variable z must be somewhere in the range delimited by the K_{low} and K_{high} numbered intervals, i.e.,

$$z \in \left[K_{\text{low}}\delta - \frac{\delta}{2}, K_{\text{high}}\delta + \frac{\delta}{2}\right] \tag{5.3}$$

we must always have

$$\sum_{k\in[K_{\text{low}},K_{\text{high}}]} \mathcal{P}[z]_k = 1 \qquad . \tag{5.4}$$

Some further notation: let

$$c(k) = k\delta \qquad \forall k \in \mathbb{Z} \tag{5.5}$$

be the center of the kth interval and

$$C(k) = \left[k\delta - \frac{\delta}{2}, k\delta + \frac{\delta}{2}\right) \qquad \forall k \in \mathbb{Z} \tag{5.6}$$

be the kth interval itself. We also let $\mathbb{P} = [0,1]^\infty$ denote the set of all possible quantized pdfs.

Given any two independent random variables z and w whose pdfs $\mathcal{P}[z]$ and $\mathcal{P}[w]$ are known, we can find the pdf of their sum, difference, or any other dyadic function of the two variables. Consider the case of their sum $z + w$. Then by the definition of the quantized pdf, we have

$$\mathcal{P}[z+w]_i = P[z+w \in C(i)] \qquad . \tag{5.7}$$

To evaluate these probabilities exactly we would need the details of the shape of the pdfs of z and w inside each little interval $C(i)$. However, for small δ, we can approximate things by assuming all the probability density in each interval of each pdf is at the center of the interval $c(i)$ (i.e., we are approximating $p_z(z)$ as

$$p_z(z) \approx \sum_i \mathcal{P}[z]_i \delta(z - c(i)) \tag{5.8}$$

and similarly for $p_w(w)$. Given this, and the assumption that z and w are independent, we have

$$\mathcal{P}[z+w]_i \approx \sum_{(j,k):c(j)+c(k)\in C(i)} \mathcal{P}[z]_j \mathcal{P}[w]_k \qquad . \tag{5.9}$$

We can handle other dyadic functions of random variables similarly, e.g.,

$$\mathcal{P}[z-w]_i \approx \sum_{(j,k):c(j)-c(k)\in C(i)} \mathcal{P}[z]_j \mathcal{P}[w]_k \tag{5.10}$$

and

$$\mathcal{P}[\max(z,w)]_i \approx \sum_{(j,k):\max(c(j),c(k))\in C(i)} \mathcal{P}[z]_j \mathcal{P}[w]_k \quad . \tag{5.11}$$

We shall find it convenient to define operators combining a pair of pdfs, e.g., the $\widetilde{+}$ operator combines $\mathcal{P}[z]$ and $\mathcal{P}[w]$ as follows:

$$\mathcal{P}[z]\widetilde{+}\mathcal{P}[w] = \mathcal{P}[z+w] \tag{5.12}$$

and similarly we can define $\widetilde{-}$, $\widetilde{\max}$, etc., in the obvious fashion.

5.2. Analyzing LDPC Code Decoding with Density Evolution

We are now ready to start analyzing the behavior of the pdfs of the variables in the belief propagation decoder as the decoder proceeds through its iterations. We start by assuming, without loss of generality, that the transmitted codeword is the all-zero codeword. This is valid since, given any codeword $\mathbf{x}$ satisfying (3.1), we can set up a one-to-one equivalence between the behavior pattern of the LDPC decoder with *a priori* likelihoods $L(p_i)$ and the behavior of the decoder with inputs $(-1)^{x_i}L(p_i)$. To see this, consider the various steps of the log version of the LDPC decoder (i.e., (3.32), (3.33), (3.34)). Suppose we replace our initial $L(p_i)$ values with

$$\widehat{L(p_i)} = (-1)^{x_i}L(p_i) \quad . \tag{5.13}$$

Then in the first step of the algorithm instead of the original $L(q_{ji})$ we get

$$\widehat{L(q_{ji})} = \widehat{L(p_i)} = (-1)^{x_i} L(q_{ji}) \tag{5.14}$$

Now consider (3.32). Instead of the original $L(r_{ji})$ we now get

$$\begin{aligned} \widehat{L(r_{ji})} &= 2\tanh^{-1}\left(\prod_{k\in N(j)\backslash i} \tanh(\widehat{L(q_{jk})}/2)\right) \\ &= 2\tanh^{-1}\left(\prod_{k\in N(j)\backslash i} (-1)^{x_k}\tanh(L(q_{jk})/2)\right) \quad . \end{aligned} \tag{5.15}$$

But since $\mathbf{x}$ is a valid codeword, the set of all the x_i bits that participate in any check must have even parity, so

$$\prod_{k\in N(j)} (-1)^{x_k} = 1 \tag{5.16}$$

and hence

$$\prod_{k\in N(j)\backslash i} (-1)^{x_k} = (-1)^{x_i} \tag{5.17}$$

so we have

$$\begin{aligned} \widehat{L(r_{ji})} &= (-1)^{x_i} 2\tanh^{-1}\left(\prod_{k\in N(j)\backslash i} \tanh(L(q_{jk})/2)\right) \\ &= (-1)^{x_i} L(r_{ji}) \quad . \end{aligned} \tag{5.18}$$

But given that, the next iteration's set of $L(q_{ji})$ and $L(q_i)$ will, given the new inputs, become

$$\begin{aligned}\widehat{L(q_{ji})} &= \widehat{L(p_i)} + \sum_{k \in M(i) \backslash j} \widehat{L(r_{ki})} \\ &= (-1)^{x_i} L(p_i) + (-1)^{x_i} \sum_{k \in M(i) \backslash j} L(r_{ki}) \\ &= (-1)^{x_i} L(q_{ji}) \end{aligned} \tag{5.19}$$

and similarly

$$\widehat{L(q_i)} = (-1)^{x_i} L(q_i) \quad . \tag{5.20}$$

Hence, given any set of initial *a priori* likelihoods $L(p_i)$ with known decoding behavior, and some codeword $\mathbf{x}$, we can find an equivalent set of initial likelihoods $\widehat{L(p_i)}$ whose decoding behavior is identical to that of the $L(p_i)$ except that all messages involving bit i are multiplied by $(-1)^{x_i}$. If the original set of likelihoods corresponded to the transmission of codeword $\mathbf{x}$ and decoded successfully to the codeword $\mathbf{x}$, then

$$\text{sgn}(L(q_i)) = (-1)^{x_i} \tag{5.21}$$

and hence the new likelihood values would obey

$$\text{sgn}(\widehat{L(q_i)}) = (-1)^{x_i}(-1)^{x_i} = (-1)^0 \tag{5.22}$$

and hence would decode to the all-zero codeword. Hence, given the situation that any codeword $\mathbf{x}$ was transmitted, we can find an equivalent situation (an equivalent set of likelihoods) corresponding to the case of the all-zero codeword. We can hence assume the codeword is all-zeros without loss of generality, as we said earlier.

We are now ready to see how to model the behavior of the LDPC decoder by looking at probability densities of the various variables involved. Let us define a

bit of notation to simplify things. We shall, following [15], call the messages from variable nodes to check nodes v_k for some $k = 1, 2, \ldots$. This means that each v_k is equal to $L(q_{ji})$ for some pair j, i; we assign indices to each of these messages in an arbitrary fashion. Similarly let the u_k be the various messages $L(r_{ji})$ from check nodes to variable nodes. We assume that these v_k and u_k variables are all independent random variables. (This is not, strictly speaking, the case, but to a good approximation we can assume it to be true as we consider larger and larger networks of nodes, in the limit as $N \to \infty$.) We also assume that the nodes are connected randomly, with given densities λ_i telling what fraction of variable nodes have i check node neighbors and ρ_i telling what fraction of check nodes have i variable node neighbors. We also assume that our initial *a priori* likelihoods $L(p_i)$ are independent with pdf $\mathcal{P}[L(p)]$.

Suppose we know the pdf $\mathcal{P}[u]$, which we assume is the pdf of any random one of the u_k variables. We wish to find $\mathcal{P}[v]$, the pdf of some random v_k, i.e., the message from some random variable node. If v_0 is the output of a variable node of degree i receiving messages $u_0, u_1, u_2, \ldots, u_{i-1}$ from its i check node neighbors, and if we assume that v_0 is the message going back to the same check node that gave us the u_0 message, we have by (3.33)

$$v_0 = L(p) + \sum_{k=1}^{i-1} u_k \tag{5.23}$$

where $L(p)$ is the *a priori* likelihood associated with that variable node. Since we have assumed the u_k all have pdf $\mathcal{P}[u]$, and are independent, we must have

$$\mathcal{P}[v_0] = \mathcal{P}[L(p)] \tilde{+} \tilde{+}^{i-1} (\mathcal{P}[u]) \tag{5.24}$$

where the iterated operator on pdfs $\widetilde{+}^i$ is defined in the obvious fashion as

$$\widetilde{+}^i(\mathcal{P}[x]) = \begin{cases} \mathcal{P}[x] & \text{if } i = 1 \\ \mathcal{P}[x] \widetilde{+} \left(\widetilde{+}^{i-1}(\mathcal{P}[x])\right) & \text{if } i > 1 \end{cases} \quad . \tag{5.25}$$

Since we wish to find the pdf of any random v of any random degree, we have to average over all possible degrees i, weighting by the probability λ_i that we are dealing with a variable node of degree i. This gives us

$$\begin{aligned} \mathcal{P}[v] &= \sum_{i=2}^{\infty} \lambda_i \mathcal{P}[L(p)] \widetilde{+} \widetilde{+}^{i-1}(\mathcal{P}[u]) \\ &= \mathcal{P}[L(p)] \widetilde{+} \sum_{i=2}^{\infty} \lambda_i \widetilde{+}^{i-1}(\mathcal{P}[u]) \end{aligned} \tag{5.26}$$

where addition and scalar multiplication of pdfs are done in the obvious (point-by-point) way. Note that since $\sum_i \lambda_i = 1$, the sums above preserve the property that the pdfs are normalized, i.e., that for a variable x,

$$\sum_{\ell=-\infty}^{\infty} \mathcal{P}[x]_\ell = 1 \quad . \tag{5.27}$$

We can make our equation simpler by define a new function $\lambda : \mathbb{P} \to \mathbb{P}$ on pdfs

$$\lambda(\mathcal{P}[x])_\ell = \sum_{i=2}^{\infty} \lambda_i \left(\widetilde{+}^{i-1} \mathcal{P}[x]\right)_\ell \tag{5.28}$$

which gives us

$$\mathcal{P}[v] = \mathcal{P}[L(p)] \widetilde{+} \lambda(\mathcal{P}[u]) \quad . \tag{5.29}$$

We now compute the pdfs of messages coming out of the check nodes. Consider a check node of degree i with an output u_0 and inputs from its variable node neighbors

$v_0, v_1, \ldots, v_{i-1}$. From (3.67) we have

$$u_0 = \mathop{\mathcal{R}}_{k=1}^{i-1} v_k \tag{5.30}$$

where the function $\mathcal{R}(a,b)$ is as defined in (3.66). Note that, as we explained in Chapter 3, in the case of max-product decoding we use the function $\mathcal{M}(a,b)$ instead of $\mathcal{R}(a,b)$; this is the only difference between the max-product and sum-product variants of the LDPC decoder. Anyway, we can now derive the relationship between the pdfs of u_0 and the v_i:

$$\mathcal{P}[u_0] = \widetilde{\mathcal{R}}^{i-1}\mathcal{P}[v] \tag{5.31}$$

and, as before, generalize to the case of nodes of random degrees:

$$\mathcal{P}[u] = \sum_{i=2}^{\infty} \rho_i \widetilde{\mathcal{R}}^{i-1}\mathcal{P}[v] \quad . \tag{5.32}$$

As before, we can define a function on pdfs $\rho : \mathbb{P} \to \mathbb{P}$ as

$$\rho\left(\mathcal{P}[x]\right)_\ell = \sum_{i=2}^{\infty} \rho_i \left(\widetilde{\mathcal{R}}^{i-1}\mathcal{P}[x]\right)_\ell \tag{5.33}$$

which lets us rewrite $\mathcal{P}[u]$ as

$$\mathcal{P}[u] = \rho(\mathcal{P}[v]) \quad . \tag{5.34}$$

We need one final pdf relationship: we need the pdf of the pseudo-posterior probabilities $L(q_i)$. We can find $\mathcal{P}[L(q)]$ as follows: since the pseudo-posterior probability associated with a variable node is just the message v_i from that node to a check node neighbor, **plus** the check-to-variable message that we did not include in

the computation of v_i, we have that

$$L(q_i) = v_i + u_i \tag{5.35}$$

where v_i is the message from the variable node to a check node, and u_i is the previous message from that check node to the variable node. Hence, we have

$$\mathcal{P}[L(q)] = \mathcal{P}[v] \widetilde{+} \mathcal{P}[u] \quad . \tag{5.36}$$

Since we assumed an all-zero codeword, the probability that the decoder has decoded a given bit incorrectly is just

$$P[L(q) < 0] = \left(\sum_{i=-\infty}^{-1} \mathcal{P}[L(q)]_i \right) + \frac{\mathcal{P}[L(q)]_0}{2} \tag{5.37}$$

(the $\mathcal{P}[L(q)]_0/2$ term is there because the interval $C(0)$ is half below and half above zero).

We are now ready to do density evolution to estimate the probability of error for decoding an LDPC code of given λ_i, ρ_i. We do this via a repeated iteration of computing pdfs analogous to the repeated iteration of the steps of the LDPC decoder. We proceed as follows:

1. Initialize the pdf of check-node messages as

$$\mathcal{P}[u]_\ell = \delta_{\ell,0} \tag{5.38}$$

which corresponds to the messages u_i being identically zero. (This is not quite the same as the initialization

$$L(q_{ji}) = L(p_i) \tag{5.39}$$

we used back in our description of the algorithm in Chapter 3, but initializing all the check-node messages $L(r_{ji}) = 0$ leads to $L(q_{ji}) = L(p_i)$ after the next application of (3.33).)

2. Compute $\mathcal{P}[v]$ from (5.29).

3. Compute $\mathcal{P}[L(q)]$ from (5.36).

4. Compute the probability of error $P_e = P[L(q) < 0]$ from (5.37).

5. Compute the next $\mathcal{P}[u]$ from (5.34).

6. Go back to Step 1 and repeat for as many iterations as one's LDPC decoder is set to run.

5.3. Computing Thresholds

As mentioned in the beginning of this chapter, we often want to compute a so-called threshold value for a given LDPC code; the higher the threshold value, the higher the noise in the channel has to be before we start getting errors in the decoder. Here we explain how to compute thresholds for LDPC codes used over AWGN channels.

First, in order to do any performance evaluation of LDPC codes over AWGN channels, we need to figure out what the pdf of the initial likelihood values $\mathcal{P}[L(p)]$ is. Assume that our bits x_i are converted to bipolar form and sent over an AWGN channel with noise variance σ^2, i.e. the channel output is

$$y_i = (-1)^{x_i} + n_i \qquad n_i \sim N(0, \sigma^2) \qquad . \tag{5.40}$$

By the definition of the log-likelihood ratio, we have

$$\begin{aligned} L(p_i) &= \log \frac{P(x_i = 0|y_i)}{P(x_i = 1|y_i)} \\ &= \log \frac{P(y_i|x_i = 0)}{P(y_i|x_i = 1)} \\ &= \log \frac{\exp(-(y_i - 1)^2/(2\sigma^2))}{\exp(-(y_i + 1)^2/(2\sigma^2))} \\ &= -\frac{(y_i - 1)^2}{2\sigma^2} + \frac{(y_i + 1)^2}{2\sigma^2} \\ &= \frac{2y_i}{\sigma^2} \end{aligned} \tag{5.41}$$

Now if we transmit an all-zeros codeword, then $y_i \sim N(1, \sigma^2)$, so we have that $L(p_i)$ is Gaussian too, with

$$L(p_i) \sim N\left(\frac{2}{\sigma^2}, \frac{4}{\sigma^2}\right) \quad . \tag{5.42}$$

Given the above pdf, we can use the density evolution algorithm to estimate the probability of error for an LDPC code with given parameters λ_i, ρ_i at any noise level σ^2. We define the threshold as the value of σ^2 for which the probability of error is 10^{-6}. This definition is somewhat arbitrary (we could have picked another value other than 10^{-6}), but the exact definition turns out not to matter too much. Chung [15] uses a slightly different definition, but we find this one somewhat easier to compute with; to find the threshold, we do a binary search on values of σ^2 until we find one which leads to a P_e of approximately 10^{-6}, and take that as our threshold.

Chapter 6

Computing Soft BER Estimates

In recent years, there has been a great deal of interest in two closely related types of powerful error-correcting codes: turbo codes [14], an low-density parity-check (LDPC) codes [5], [6]. Due to the complexity of decoding LDPC codes or turbo codes, those who wish to evaluate the performance of their codes are faced with the problem that Monte Carlo simulations become more and more computationally expensive if one is interested in the behavior of the code for lower and lower bit error rates (BER). Recently Hoeher *et al.* [18] suggested a way to handle this problem, through the use of what one might call "soft error estimates" or "soft BER estimates" which allow the estimation of bit error rates in simulations where the traditional Monte Carlo computation of BER gives too few simulated bit errors to be of use; Hoeher also discussed their applicability to the simulation of turbo codes. In this chapter we discuss the applicability of Hoeher's method and other soft error estimation methods to performance evaluation of LDPC codes on AWGN channels. We present theoretical reasons why the Hoeher method should work well in this situation, but find problems with the technique in actual use.

6.1. Soft Error Estimates

Consider the typical case of a turbo or LDPC code being used over a memoryless channel: we send coded bipolar bits $U_i \in \{-1, 1\}$ over the channel and then use the received channel outputs y_i to compute initial log-likelihood ratios

$$L_{\text{init,i}} = \log \frac{P(U_i = +1|y_i)}{P(U_i = -1|y_i)} \quad . \tag{6.1}$$

Then the decoding algorithm operates iteratively on the log-likelihood values to get final log-likelihood values for each bit. In traditional Monte Carlo simulations one

hard-decodes the L_i output from the decoding algorithm as:

$$\hat{U}_i = \text{sgn}(L_i) \quad . \tag{6.2}$$

and then compares the hard decoded bits $\hat{U}_i$ against the original input bits U_i; the estimated BER is the fraction of $\hat{U}_i$ that do not match the original U_i. Define $W_i = 1 - \delta(U_i, \hat{U}_i)$ where W_i is 1 if there is an error at bit i; then one estimates the BER by

$$P_{e(\text{hard})} = \bar{W} = \frac{1}{N} \sum_{i=1}^{N} W_i \tag{6.3}$$

where $\bar{W}$ is the arithmetic mean of the W_i.

In [18], Hoeher *et al.* noted that since L_i is the decoder's estimate of the log-likelihood ratio between the probabilities of the symbol being $+1$ or -1, then an estimate of the probability of error at bit i is

$$Z_i = \frac{1}{1 + \exp(|L_i|)} \tag{6.4}$$

and one can estimate the overall BER by averaging the Z_i gathered from a group of simulations of the decoder:

$$P_{e(\text{soft})} = E[Z] \approx \bar{Z} = \frac{1}{N} \sum_{i=1}^{N} Z_i \tag{6.5}$$

where $\bar{Z}$ is the arithmetic mean of the observed Z_i samples.

This results in a soft error estimate, as opposed to the "hard error estimate" from hard-decision simulation. This technique was called Method 2 in [18].

This procedure intuitively seems like it should give a good estimate of the BER. In [18], Hoeher *et al.* proved that this is a valid estimate when used on the initial L_i (i.e., for the uncoded channel) and asserts that this is the case for the L_i after

turbo decoding. We now look into the case of the L_i outputs from LDPC decoding.

6.2. Density Evolution and LDPC Decoding

As mentioned in Chapter 5, the density evolution method of Richardson and Urbanke [17], [15] is a powerful technique for modeling the behavior of the LDPC decoding algorithm. For any given class of LDPC codes, it provides a ways to estimate the probability density function of the sum-product decoding L_i values under the assumption that the all-+1 codeword is sent. (Assuming a symmetric channel, we can assume this without loss of generality, as for the case when -1 is sent the L_i distribution will just be the appropriate mirror image.)

In [15], [19], it is argued that for LDPC sum-product decoding with an AWGN channel the L_i are, to a very good approximation, Gaussian random variables with the property that their variance is twice the mean, i.e., $L_i \sim N(m_i, 2m_i)$ for some m_i. Given this, we can readily show that

$$\begin{aligned} E[Z_i] &= \frac{1}{\sqrt{4\pi m_i}} \int_{-\infty}^{\infty} \frac{1}{1+\exp(|x|)} \exp\left(-\frac{(x-m_i)^2}{4m_i}\right) dx \\ &= \frac{1}{\sqrt{4\pi m_i}} \int_{0}^{\infty} \frac{1}{1+\exp(x)} \exp\left(-\frac{(x-m_i)^2}{4m_i}\right) dx \\ &\quad + \frac{1}{\sqrt{4\pi m_i}} \int_{0}^{\infty} \frac{1}{1+\exp(x)} \exp\left(-\frac{(x+m_i)^2}{4m_i}\right) dx \end{aligned} \tag{6.6}$$

and hence

$$\begin{aligned} E[Z_i] &= \frac{1}{\sqrt{4\pi m_i}} \int_{0}^{\infty} \exp\left(\frac{(-x^2-m_i^2)}{4m_i}\right) \exp\left(-\frac{x}{2}\right) dx \\ &= \frac{1}{\sqrt{4\pi m_i}} \int_{0}^{\infty} \exp\left(-\frac{(x+m_i)^2}{4m_i}\right) dx \\ &= \frac{1}{\sqrt{4\pi m_i}} \int_{-\infty}^{0} \exp\left(-\frac{(x-m_i)^2}{4m_i}\right) dx \\ &= P_L(L_i < 0) = P_e \end{aligned} \tag{6.7}$$

and hence that estimating $E[Z_i]$ gives an estimate of the probability of error.

As it happens, one can generalize the previous proof and show the validity of $E[Z_i]$ as an estimate of P_e without the assumption of Gaussian distributions $N(m_i, 2m_i)$ for the likelihood ratios, which, as mentioned in [15] and [19], is only a good approximation anyway. All one really needs is the lesser condition of exponential symmetry, i.e., that the pdf of the L_i, $p_L(\ell)$ satisfies

$$p_L(\ell) = \exp(\ell) p_L(-\ell) \quad . \tag{6.8}$$

This condition is true for the initial log-likelihood values from the AWGN channel; since the initial values are Gaussian, this is equivalent to the condition that the variance of the log-likelihood values is twice the mean. In [17], Richardson and Urbanke show that exponential symmetry is **invariant** under iterations of sum-product decoding, so if the initial log-likelihood values have exponential symmetry, so will the final values.

Given exponential symmetry for the final log-likelihood values L_i, we have

$$\begin{aligned} E[Z_i] &= \int_{-\infty}^{\infty} \frac{1}{1+\exp(|\ell|)} p_L(\ell) d\ell \\ &= \int_0^{\infty} \frac{1}{1+\exp(\ell)} \left(p_L(\ell) + p_L(-\ell)\right) d\ell \\ &= \int_0^{\infty} \frac{1}{1+\exp(\ell)} \left(\exp(\ell) p_L(-\ell) + p_L(-\ell)\right) d\ell \\ &= \int_0^{\infty} p_L(-\ell) d\ell \\ &= P(L_i < 0) = P_e \end{aligned} \tag{6.9}$$

and hence whenever we have exponential symmetry, $E[Z_i] = P_e$ and the soft error estimates of $E[Z_i]$ are still valid estimates of the bit error rate. Hoeher *et al.* [18] give a similar proof for another BER estimation technique, which they call Method 3 and which is closely related to the Method 2 we examine here.

Given that the technique of density evolution is available to us, the reader may be wondering why one would bother with simulations at all, with or without soft error estimates, if one can directly compute pdfs of the L_i and expected P_e values for any given SNR? The answer lies in the fact that density evolution only applies to a class of LDPC codes; if one is interested in the behavior of a specific code of a specific given length, one has no choice but to simulate the code or try to use some sort of bounding technique. Also, the density evolution techniques, strictly speaking, apply only in the limit of infinite code length; they neglect the effects of finite code length or of four-cycles in the code, which can greatly effect the performance of LDPC codes.

6.3. BER Variances

We have presented two different methods of estimating the BER for a system employing LDPC codes, the traditional hard estimate $P_{e(\text{hard})} = \bar{W}$ in (6.3) and the soft estimate $P_{e(\text{soft})} = \bar{Z}$ in (6.5). The question naturally arises as to just how good either of these estimates actually is in practice. To answer the question, we look at the variances of $\bar{W}$ and $\bar{Z}$.

From the standard Bernoulli distribution, we can easily compute

$$Var[W_i] = P_e(1 - P_e) \quad . \tag{6.10}$$

Note that for most common values of P_e we can approximate $Var[W_i]$ by P_e without difficulty. The variance of Z_i is defined as

$$Var[Z_i] = E[Z_i^2] - E[Z_i]^2 \quad . \tag{6.11}$$

This is difficult, if not impossible, to compute analytically from the assumed Gaus-

sian distribution of the L_i but can be computed numerically (as we do below in a couple of examples). Alternately, if one is doing a simulation of an LDPC system with soft error estimates, one can take the observed Z_i values and compute the observed sample variance and take that as an estimate of $Var[Z_i]$.

With either the hard or soft error estimates, given the variance of the individual values ($Var[W_i]$ or $Var[Z_i]$), we can compute the variance of the final estimates ($Var[\bar{W}]$ or $Var[\bar{Z}]$). Assuming that the W_i and Z_i are independent and identically distributed, we have

$$Var[\bar{W}] = \frac{Var[W_i]}{N} = \frac{P_e(1-P_e)}{N} \tag{6.12}$$

and

$$Var[\bar{Z}] = Var[Z_i]/N \quad . \tag{6.13}$$

It is reasonable to measure the "goodness" of one's BER estimate by comparing the squared estimate to the variance. One can define signal-to-noise ratio (SNR) values on the estimates as follows:

$$\mathrm{SNR}_{\mathrm{soft}} = 10\log_{10}\left(\frac{\bar{Z}^2}{Var[\bar{Z}]}\right) \tag{6.14}$$

and similarly define $\mathrm{SNR}_{\mathrm{hard}}$ based on $\bar{W}$. The higher these SNRs are, the better the estimate.

6.4. LDPC Max-Product Decoding

Often, for efficiency reasons, one decodes LDPC codes with the computationally less expensive max-product algorithm instead of the sum-product algorithm. The question that comes to mind immediately is: Does the $E[Z_i]$ soft error estimate still have any validity in the case of max-product decoding? Our previous proof depends

on the exponential symmetry property, which is **not** preserved by max-product decoding; in fact, **no** algorithm for LDPC decoding other than the sum-product one can preserve exponential symmetry [15]. Given this, let us try examining some sample code classes under max-product density evolution and see how P_e compares to $E[Z_i]$ for these codes.

Table 6.1 shows the P_e and $E[Z_i]$ values computed for various values of the AWGN noise variance σ^2 via max-product density evolution for a rate 1/2, regular, column weight 3 LDPC code. We see that the soft error estimates are of the same order of magnitude as the true P_e values, with the soft error estimates being consistently low by a factor of about 0.6. This deviation is due to the likelihoods from the max-product decoding deviating from exponential symmetry; the results are still approximately Gaussian, but the variance, instead of being twice the mean, is a bit over 3 times the mean. In principle, if the amount of deviation from exponential symmetry (i.e. the ratio $Var[L]/E[L]$) were known, one might be able to modify the soft error estimates to compensate for this. A relatively straightforward modification of (6.6) for the case where $L \sim N(m, km)$ (k not necessarily equal to two) shows that replacing Z_i with

$$\hat{Z}_i = \frac{1}{1 + \exp(2|L_i|/k)} \tag{6.15}$$

results in an $E[\hat{Z}_i]$ equal to P_e.

Table 6.2 shows similar results done with the parameters from a rate 0.94 column weight 3 code used in [9]. Here the soft error results are low by a factor of about 0.8 compared to the true P_e.

Table 6.1: Max-product P_e vs. $E[Z_i], Var[Z_i]$ results for rate 1/2 code

σ^2	P_e	$E[Z_i]$	$Var[Z_i]$
0.675500	1.31883e-05	8.1168e-06	2.01004e-06
0.675510	0.000388843	0.000240111	5.97675e-05
0.675520	0.00325519	0.00201763	0.000503232
0.675530	0.0124999	0.00777371	0.00192095
0.675540	0.0296471	0.0184918	0.00445555
0.675550	0.052169	0.0326394	0.00756955
0.675560	0.0762037	0.0478456	0.0106113
0.675570	0.0989503	0.0623723	0.0132178
0.675580	0.119094	0.075378	0.0153027

Table 6.2: Max-product P_e vs. $E[Z_i], Var[Z_i]$ results for rate 0.94 code

σ^2	P_e	$E[Z_i]$	$Var[Z_i]$
0.189900	2.98467e-07	2.57098e-07	5.85325e-08
0.189910	0.00372522	0.0031545	0.000731127
0.189920	0.0417915	0.0338486	0.00737187
0.189930	0.0900744	0.0704572	0.0138307
0.189940	0.123619	0.0949367	0.0172159

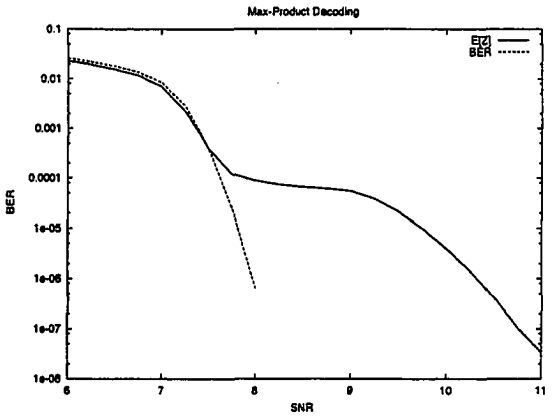


Figure 6.1: Max-product decoding simulation results.

6.5. Experimental Results

Here we present some results from simulations of the rate 0.94 LDPC code on the AWGN channel, with curves plotted of both the BER (i.e. P_e) and the soft error estimate $E[Z]$. Fig. 6.1 shows a plot of BER and $E[Z]$ vs. SNR for the case of max-product decoding; Fig. 6.3 shows similar results for sum-product decoding. We also examine the variances of these estimates by plotting the SNRs for both the hard and soft estimates as defined in (6.14). The plot for the sum-product case is given in Fig. 6.4; similarly, the plot for the max-product simulations is given in Fig. 6.2.

In the sum-product simulation case we see in the figures reasonably good agreement between the hard and the soft error estimates down to input SNRs of 7.5 dB. This corresponds to the behavior we see on the SNR graphs where the SNR for the hard estimate starts going down markedly around 7.5 dB. We see similar behavior for the max-product decoding simulations around input SNRs of 7.75 dB. With the soft error estimate SNR curves, we see the SNR start to decline at around 9.5 dB for the sum-product case and around 9 dB. If we set a threshold of 20 dB as the limiting estimate SNR below which we no longer consider our estimates to be useful

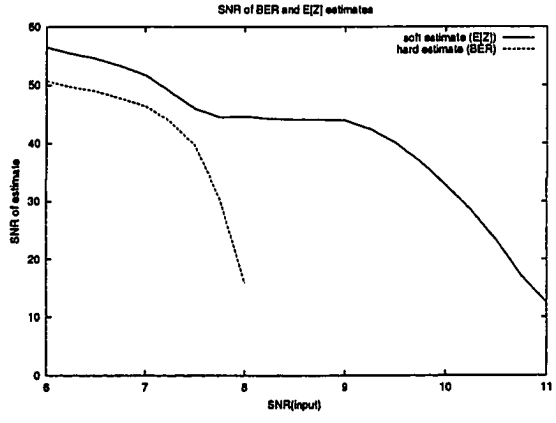


Figure 6.2: Max-product decoding simulation SNR values.

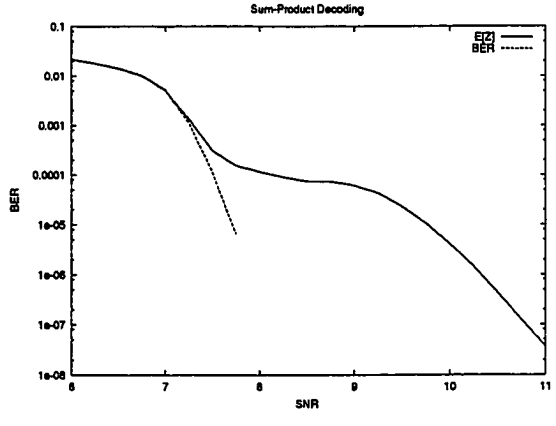


Figure 6.3: Sum-product decoding simulation results.

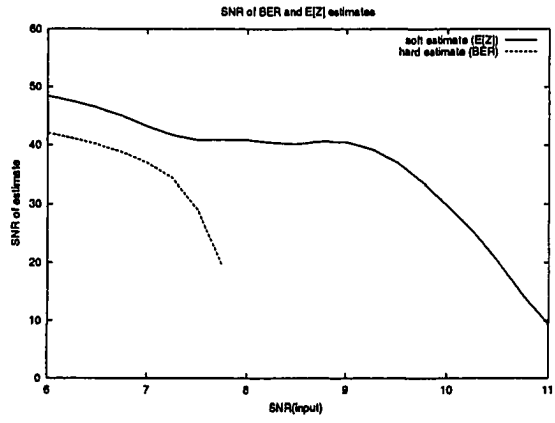


Figure 6.4: Sum-product decoding simulation SNR values.

(this corresponds, in the hard-estimate case, to the traditional rule-of-thumb that one wants at least 100 bit errors to appear in one's simulations), we see that in the sum-product case we hit that threshold at 7.75 dB input SNR for the hard estimates, and 10.5 dB input SNR for the soft estimates. Similarly, for max-product decoding we hit that threshold at around 7.8 dB for hard estimates and around 10.6 dB for soft estimates. Just looking at the SNR curves would lead us to expect that the soft estimates were valid out to this point (10.5 dB for sum-product, 10.6 dB for max-product). However, the peculiar shape of the $E[Z]$ curves leads us to suspect that these soft estimates are not, in fact, valid throughout this range and, in fact, start having problems even before the hard estimates become unreliable.

6.6. What Went Wrong?

Obviously something is amiss in the reasoning that lead us to the use of these soft estimates $E[Z]$. The proofs above used one of two assumptions, that the L values were Gaussian with variance twice the mean, or that the L pdf had exponential symmetry. As we show by examining graphs derived from the L values from various LDPC max-product decoder runs, these assumptions do not seem to be valid for

the actual decoder outputs. Figs. 6.5, 6.6, and 6.7 show histograms of the observed L values from simulation runs of max-product decoding of the aforementioned rate 0.94 LDPC code at SNRs 7dB, 8dB, and 9dB. Also plotted on the graph are the "ideal" distributions derived from fitting a Gaussian distribution to the observed L mean and variance. (Note that since the actual simulation used words with both zero and one bits, i.e., $U_i = +1$ or $U_i = -1$, our observed L distributions would actually be expected to be the sum of two Gaussians with the same variance and means at $\pm m$ for some m. The graphs below reflect this.) Note that the observed L distributions look roughly Gaussian, but with some noticeable deviations from the Gaussian ideal. Chi-square tests confirm this; Table 6.3 shows results of chi-square tests against the Gaussian assumption. The table shows that the observed L values fail the test of Gaussianity, and do so overwhelmingly. Similar results for sum-product decoding are shown in Table 6.4.

As for the assumption of exponential symmetry, that seems not to hold either. Fig. 6.8 shows a plot of the observed ratio of pdf values $p(L)/p(-L)$ versus the value expected from exponential symmetry, $\exp(L)$, for the L values observed from max-product decoding of our code at SNR 7dB. As we can see, the observed ratio deviates noticeably from the exponential symmetry condition. (To avoid some minor complications resulting from the presence of L values corresponding to both original zero and one codeword bits in our distribution, this graph was computed using just the L values corresponding to zero bits. A similar test with just the $-L$ values from the one bits produced an essentially identical graph.)

6.7. Alternate Methods for Soft Error Estimation

In this section we consider some alternate techniques for soft error estimation, and attempt to apply them to the same max-product LDPC decoding situation. As we

Table 6.3: Chi-square test of Gaussianity for L values from max-product decoding

SNR (dB)	Degrees of freedom	χ^2
7	21	156632
7.5	85	175633
8	85	203034
8.5	85	158901
9	85	114304
9.5	85	516728
10	85	1.18226×10^6
10.5	85	4.48140×10^6
11	85	10655.8
12	85	8742.39
13	85	8755.16

Table 6.4: Chi-square test of Gaussianity for L values from sum-product decoding

SNR (dB)	Degrees of freedom	χ^2
7	5	2126.73
7.5	62	110532
8	85	581551
8.5	85	227562
9	85	91071.7
9.5	85	525756
10	85	1.17160×10^6

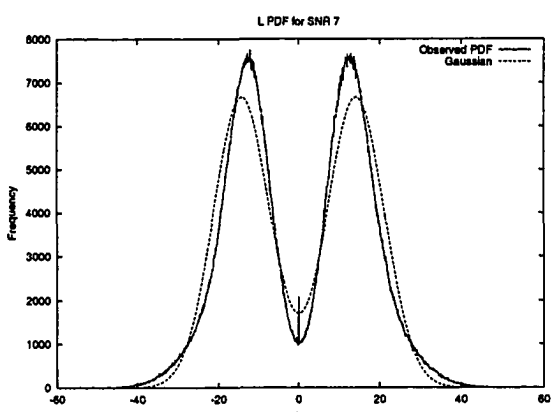


Figure 6.5: Histograms of observed L distributions versus ideal dual-Gaussian distribution for max-product decoding at 7dB SNR.

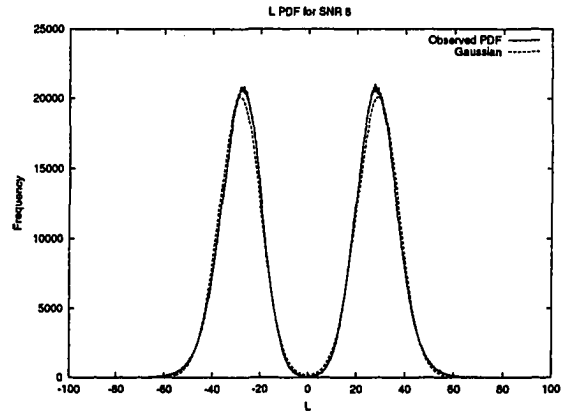


Figure 6.6: Histograms of observed L distributions versus ideal dual-Gaussian distribution for max-product decoding at 8dB SNR.

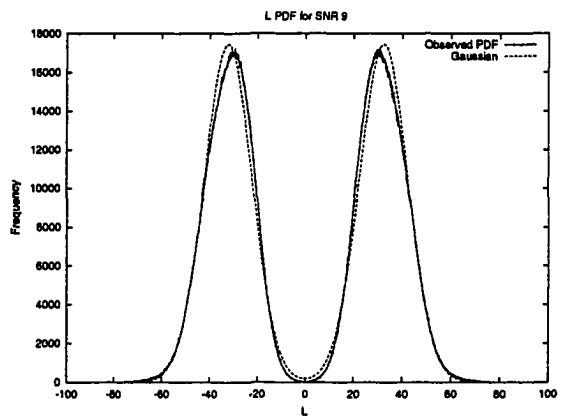


Figure 6.7: Histograms of observed L distributions versus ideal dual-Gaussian distribution for max-product decoding at 9dB SNR.

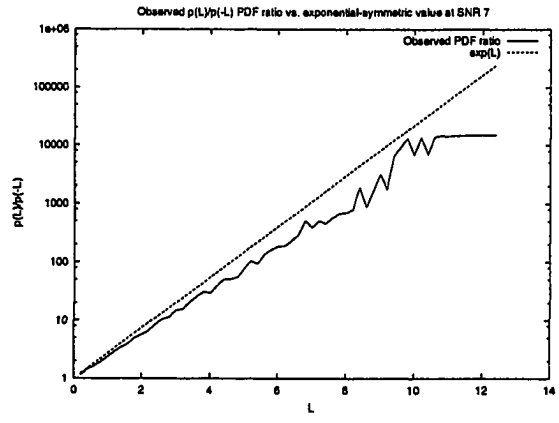


Figure 6.8: Plot of p(L)/p(-L) showing (lack of) exponential symmetry in the pdf of L.

shall see, the results are not a noticeable improvement over the results from the Hoeher method.

6.7.1. Generalized Gaussian Distributions and Asymmetric Generalized Gaussian Distributions

Since our assumption that the L values are well described by Gaussian distributions was shown by the chi-square test to be invalid, perhaps modeling the L pdf by another distribution will provide more useful results. At the same time, since the observed distributions look somewhat similar to Gaussian distributions, we should pick distributions which also look this way. In this case, we attempt to model the L distribution with so-called **generalized Gaussian distribution** (GGD) [20], [21] or **asymmetric generalized Gaussian distribution** (AGGD) [22]. Both these distributions reduce to the Gaussian distribution with appropriate choices of parameters. In addition, since Yang *et al.* [21] had some success with using generalized Gaussian distributions to model the likelihood values in turbo decoding, we hoped that similar techniques would be helpful in the LDPC decoding case. The

generalized Gaussian distribution is characterized by parameters μ, α, σ and has the form

$$f(x;\mu,\sigma,\alpha) = \frac{\alpha\eta(\alpha)}{2\sigma\Gamma(1/\alpha)} \exp\left(-|\eta(\alpha)(x-\mu)/\sigma|^{\alpha}\right) \tag{6.16}$$

where the function $\eta(\alpha)$ has the form

$$\eta(\alpha) = \sqrt{\frac{\Gamma(3/\alpha)}{\Gamma(1/\alpha)}} \quad . \tag{6.17}$$

Note that when $\alpha = 2$ this distribution reduces to the Gaussian distribution. μ and σ^2 are the mean and variance of the distribution, as usual, and α controls the broadness of the curve, and is related to the kurtosis of the distribution, which we shall call k:

$$k = \frac{E[(x-\mu)^4]}{\sigma^4} = \frac{\Gamma(5/\alpha)\Gamma(1/\alpha)}{\Gamma(3/\alpha)^2} \quad . \tag{6.18}$$

To compute a GGD-based soft error estimate given a set of observed likelihoods L_i from an LDPC decoding simulation run, one proceeds as follows:

1. "Normalize" the L_i by computing from them the values

$$\hat{L}_i = L_i U_i \quad . \tag{6.19}$$

   This amounts to flipping the sign on L_i values corresponding to one bits. By our hypothesis, the $\hat{L}_i$ values should be GGD distributed.

2. Estimate the mean μ , variance σ^2, and kurtosis k from the $\hat{L}_i$ in the obvious way.

3. Solve (6.18) to find the value of α which gives the observed kurtosis k.

4. We now have all three parameters of our GGD distribution. Our estimate of

the BER is just

$$P_{e(\text{soft,GGD})} = \int_{-\infty}^{0} f(x;\mu,\sigma,\alpha)dx \quad . \tag{6.20}$$

The asymmetric generalized Gaussian distribution [22] is a relatively straightforward extension of the GGD; instead of a single σ^2 value, there are both left and right variances σ_L^2, σ_R^2 which are estimated from the data in the obvious fashion

$$\begin{aligned} \sigma_L^2 &= \frac{1}{N_L - 1} \sum_{i=1, L_i<\mu}^{N} (L_i - \mu)^2 \\ \sigma_R^2 &= \frac{1}{N_R - 1} \sum_{i=1, L_i>\mu}^{N} (L_i - \mu)^2 \end{aligned} \tag{6.21}$$

where N_L, N_R are the number of L_i values less than/greater than μ, respectively. The pdf for the asymmetric generalized Gaussian is

$$f(x;\mu,\sigma_L,\sigma_R,\alpha) = \begin{cases} \frac{\alpha\eta(\alpha)}{(\sigma_L+\sigma_R)\Gamma(1/\alpha)} \exp\left(-|\eta(\alpha)(x-\mu)/\sigma_L|^\alpha\right) & \text{if } x < \mu \\ \\ \frac{\alpha\eta(\alpha)}{(\sigma_L+\sigma_R)\Gamma(1/\alpha)} \exp\left(-|\eta(\alpha)(x-\mu)/\sigma_R|^\alpha\right) & \text{if } x > \mu \end{cases} \quad . \tag{6.22}$$

When $\sigma_L = \sigma_R$, this distribution reduces to the GGD. As Tesei *et al.* argue in [22], although the equation (6.18) strictly speaking only holds for the GGD case, in practice it is a suitable approximation even when $\sigma_L \neq \sigma_R$ and thus can still be used to estimate the parameters of the asymmetric GGD.

Fig. 6.9 shows the results of the GGD-based soft error estimates and Fig. 6.10 shows the results of estimates based on the assumption of an asymmetric generalized Gaussian distribution. As can be seen from the plots, we still have the peculiar distortion of the curve around SNRs of 8 dB that we saw with Hoeher's method.

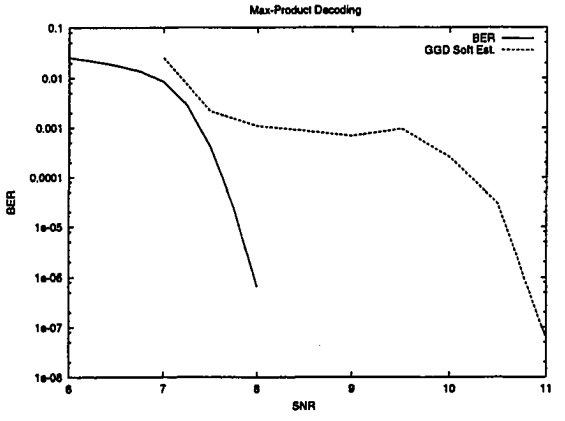


Figure 6.9: Plot of BER and generalized Gaussian distribution soft error estimate for max-product LDPC decoding.

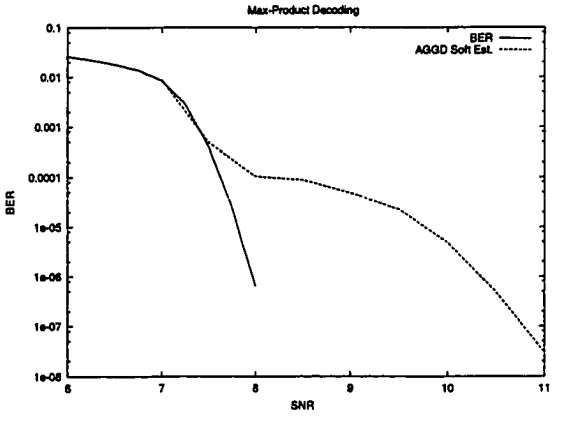


Figure 6.10: Plot of BER and asymmetric generalized Gaussian distribution soft error estimate for max-product LDPC decoding.

6.7.2. Tail Extrapolation

This technique for soft error estimates is a modification of one originally presented in Jeruchim *et al.* [23]. The idea is as follows: assume that our variable L, what Jeruchim *et al.* call the "decision variable", is GGD distributed, and assume we have on hand an estimate of the mean μ. Let us define the function

$$p(t) = P(L < \mu - t) \qquad \forall t \geq 0 \qquad . \tag{6.23}$$

Note that the probability of a bit error is $P(L < 0) = p(\mu)$. Then they show that in the limit of large t, the curve of $\log(-\log p(t))$ versus $\log t$ approaches a straight line, i.e.,

$$\log(-\log p(t)) \approx m \log t + b \qquad \text{as} \quad t \to \infty \tag{6.24}$$

for some $m, b \in \mathbb{R}$. This allows us to compute a soft error estimate as follows:

1. Estimate the pdf of the L_i for the zero bits from the histogram of the observed L_i. Compute a separate pdf for the $-L_i$ values from the one bits. (Jeruchim *et al.* [23] argue that one's decoder may perform differently for zero bits as opposed to one bits, so one should really assume that the two may have different pdfs and carry through the computation for both cases independently.

2. From the pdf for the zero-bit case, compute $p(t)$ values.

3. Perform a least-squares fit on some of the last $\log(-\log(p(t))$ versus $\log t$ pairs we have to get estimates of m, b. We are being a bit vague here as to which or how many points we wish to fit, as the user obviously has a good bit of freedom to choose here, and what choices are made may affect the quality of the result. Obviously one wants to use $p(t)$ values for t as high as possible, to be sure that we are in the linear region of the curve, but on the other hand such values are from parts of the histogram where we have relatively few data

points, so we have a trade-off here.

4. Extrapolate the line $y = mx + b$ out to $x = \log\mu$ and compute

$$P_0 = \exp(-\exp(m\log\mu + b)) \quad . \tag{6.25}$$

This is our soft error estimate for the zero bit case.

5. Repeat the whole procedure with the pdf for the one bit case to get P_1, and take our final soft error estimate to be

$$P_{esoft} = \frac{P_0 + P_1}{2} \quad . \tag{6.26}$$

Fig. 6.11 shows the performance of the tail-extrapolation soft error estimation technique; in this case three $p(t)$ points were used for the least-squares fit, starting at the point $t = \mu - L$ where the histogram counts exceeded 100. As can be seen, the curve still shows the sort of anomalies we have seen before. Perhaps looking at some actual $\log(-\log(P))$ versus $\log(t)$ curves will illuminate the situation. Fig. 6.12 has such a plot for an SNR of 7 dB. The plot shows the curve from the observed data, the least-squares line fit to the large-t end of the curve, and the point corresponding to the true BER value (i.e., at $x = \log\mu$ and $y = \log(-\log(\text{BER}))$). In this case, the tail extrapolation works well, and the extrapolated $\log(-\log(P(t)))$ hits the true BER value nicely. At an SNR of 8 dB, however, things are different, as we can see in Fig. 6.13. The least-squares line still matches up well with the large-t part of the curve we see, but the true BER is fairly far off the least-squares line, causing us to give a poor soft estimate in this case.

Perhaps trying a higher order least-squares curve fit (e.g., quadratic) will work better; after all, the assumption that the curve eventually becomes linear was based on the GGD assumption, which as we saw from the GGD estimates we attempted

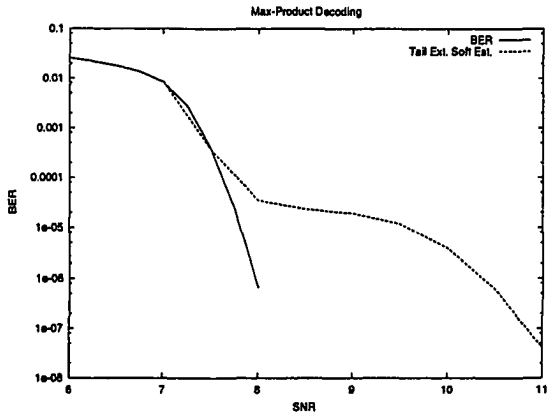


Figure 6.11: Plot of BER and tail extrapolation soft error estimate for max-product LDPC decoding.

in Figs. 6.9 and 6.10 is a rather suspect assumption. After various attempts with different choices of curve-fitting order, number of points, etc., we got Fig. 6.14, which was derived using a quadratic fit on 14 points on the curve starting at the point where the histogram count goes above 1000. The results are still not encouraging.

6.8. Conclusion

We tried various techniques for creating "soft" estimates of the bit-error-rate that would work at higher SNRs than those for which the BER from traditional Monte Carlo based simulations goes to zero. These techniques were based on various assumptions about the pdfs of the likelihood values L_i output by the LDPC decoder. Although the L_i pdf looks Gaussian, these techniques based on the assumption of the distribution being Gaussian or some sort of generalized Gaussian have failed to perform satisfactorily. The exact nature of the L_i pdf still remains a bit of a mystery, one worthy of future research.

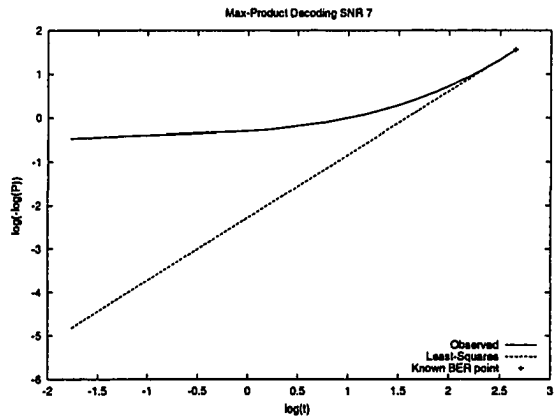


Figure 6.12: $\log(-\log(P))$ versus $\log(t)$ plot for max-product decoding at SNR 7 dB.

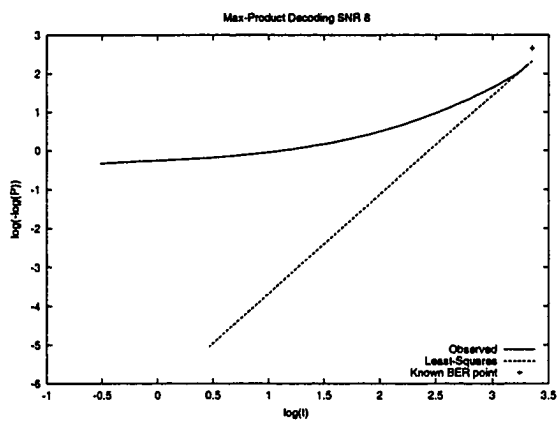


Figure 6.13: $\log(-\log(P))$ versus $\log(t)$ plot for max-product decoding at SNR 8 dB.

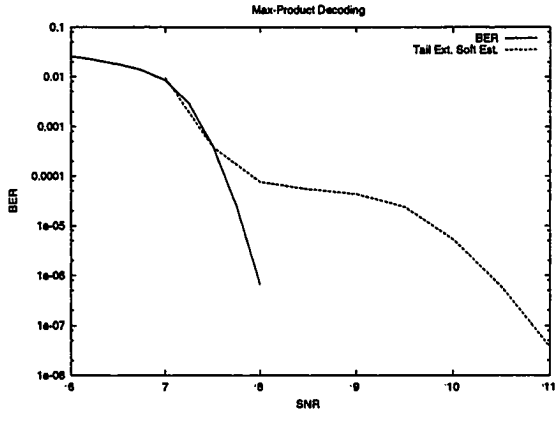


Figure 6.14: Plot of BER and quadratic tail extrapolation soft error estimate for max-product LDPC decoding.

Chapter 7

Computing Information Capacity of PR Channels

In studying the performance of codes over partial-response channels, it is helpful to have, as a basis for comparison, information-theoretic bounds on the best possible performance possible for any code when used with that channel. Such bounds are readily computable from the Shannon information capacity of the PR channel. Computing the capacity of such ISI channels has traditionally been problematic, with only loose bounds on the capacity of the channel being available. However, recently Arnold and Loeliger [24] developed a method for estimating the capacity of a PR channel with relative ease, and interestingly enough their technique involves doing simulations with a modified version of the BCJR algorithm which we presented in Chapter 2. Below we explain the Arnold-Loeliger technique and show how to estimate the capacities of PR channels and derive curves of the best possible bit-error-rate versus signal-to-noise ratio.

7.1. The Arnold-Loeliger Algorithm

Consider our standard model for the PR channel with added white Gaussian noise, as we considered it in Chapters 1 and 2, mapping inputs $x(t)$ to outputs $z(t)$:

$$z(t) = \sum_{k=0}^{L} g_k x(t-k) + N(t) \qquad N(t) \sim N(0, \sigma^2) \tag{7.1}$$

where $N(t)$ is white Gaussian noise. Define the vectors $\mathbf{z^n} = [z(0), z(1), \ldots, z(n-1)]$ and $\mathbf{x^n} = [x(0), x(1), \ldots, x(n-1)]$ for any positive integer n. One can define the entropy of these vectors in the standard way, e.g.,

$$h(\mathbf{z^n}) = \int_{\mathbf{z^n} \in \mathbb{R}^n} -p(\mathbf{z^n}) \log p(\mathbf{z^n}) d\mathbf{z^n} \tag{7.2}$$

where $p(\mathbf{z^n})$ is the joint pdf of the variables $z(0), \ldots, z(n-1)$ that constitute the vector $\mathbf{z^n}$. One can define conditional entropies similarly, e.g.

$$h(\mathbf{z^n}|\mathbf{x^n}) = \int_{\mathbf{z^n} \in \mathbb{R}^n} -p(\mathbf{z^n}|\mathbf{x^n}) \log p(\mathbf{z^n}|\mathbf{x^n}) d\mathbf{z^n} \quad . \tag{7.3}$$

We also will find useful the notion of average entropy per unit time, which we compute in the limit as n approaches infinity as

$$h(z) \equiv \lim_{n \to \infty} \frac{h(\mathbf{z^n})}{n} \tag{7.4}$$

and we may define $h(z|x)$ in the same fashion. We are interested in finding the channel capacity, how many bits of information we can get from one side of the channel to the other per unit time. The channel capacity is just the mutual information

$$I(x; z) = h(z) - h(z|x) \quad . \tag{7.5}$$

To find the channel capacity, we must be able to find both $h(z)$ and $h(z|x)$.

7.1.1. Computing $h(z|x)$

Computing $h(z|x)$ is relatively straightforward, since the conditional pdf $p(\mathbf{z^n}|\mathbf{x^n})$ turns out to be quite simple. If we are given a known vector $\mathbf{x^n}$, then the probability distribution of the $\mathbf{z^n}$ vector is completely determined by the noise vector $\mathbf{N^n} = [n(0), \ldots, n(n-1)]$. Since

$$N(t) = z(t) - \sum_{k=0}^{L} g_k x(t-k) \tag{7.6}$$

we can compute the conditional probability as

$$p(\mathbf{z}^n|\mathbf{x}^n) = p(\mathbf{N}^n) = \prod_{t=0}^{n-1} \phi(N(t)) \tag{7.7}$$

where $\phi(N)$ is the Gaussian pdf

$$\phi(N) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{N^2}{2\sigma^2}\right) \quad . \tag{7.8}$$

Since the random noise variables $N(t)$ are all independent, we find that the conditional entropy $h(\mathbf{z}^n|\mathbf{x}^n)$ is just

$$\begin{aligned} h(\mathbf{z}^n|\mathbf{x}^n) &= \int_{\mathbf{z}^n \in \mathbb{R}^n} -p(\mathbf{z}^n|\mathbf{x}^n) \log p(\mathbf{z}^n|\mathbf{x}^n) d\mathbf{z}^n \\ &= \int_{\mathbf{N}^n \in \mathbb{R}^n} -p(\mathbf{N}^n) \log p(\mathbf{N}^n) d\mathbf{N}^n \quad . \end{aligned} \tag{7.9}$$

If $n = 1$, we just have

$$\begin{aligned} h(\mathbf{z}^1|\mathbf{x}^1) &= \int_{-\infty}^{\infty} -\phi(N(0)) \log \phi(N(0)) dN(0) \\ &= \int_{-\infty}^{\infty} -\phi(N) \log \phi(N) dN \\ &= \int_{-\infty}^{\infty} \phi(N) \left[\frac{\log 2\pi\sigma^2}{2} + \frac{N^2}{2\sigma^2}\right] dN \\ &= \int_{-\infty}^{\infty} \phi(N) \frac{\log 2\pi\sigma^2}{2} dN + \frac{1}{2\sigma^2} \int_{-\infty}^{\infty} \phi(N) N^2 dN \\ &= \frac{\log 2\pi\sigma^2}{2} + \frac{1}{2\sigma^2}\sigma^2 \\ &= \frac{1 + \log 2\pi\sigma^2}{2} \quad . \end{aligned} \tag{7.10}$$

Now if $n > 1$, we have that

$$p(\mathbf{N}^n) = p(\mathbf{N}^{n-1})\phi(N(n-1)) = p(\mathbf{N}^{n-1})\phi(N) \quad . \tag{7.11}$$

For notational convenience we shortened $N(n-1)$ to just N; note that this means that $\mathbf{N}^n = [\mathbf{N}^{n-1}|N]$. We now have

$$\begin{aligned} h(\mathbf{z}^n|\mathbf{x}^n) &= \int_{\mathbf{N}^n \in \mathbb{R}^n} -p(\mathbf{N}^{n-1})\phi(N) \log \left(p(\mathbf{N}^{n-1})\phi(N)\right) d\mathbf{N}^{n-1} dN \\ &= \int_{\mathbf{N}^n \in \mathbb{R}^n} -p(\mathbf{N}^{n-1})\phi(N) \log p(\mathbf{N}^{n-1}) d\mathbf{N}^{n-1} dN \\ &\quad + \int_{\mathbf{N}^n \in \mathbb{R}^n} -p(\mathbf{N}^{n-1})\phi(N) \log \phi(N) d\mathbf{N}^{n-1} dN \quad . \end{aligned} \tag{7.12}$$

Call the two integrals in the previous equation H_1 and H_2. Now

$$\begin{aligned} H_1 &= \int_{\mathbf{N}^{n-1} \in \mathbb{R}^{n-1}} -\left(p(\mathbf{N}^{n-1}) \log p(\mathbf{N}^{n-1})\right) \int_{-\infty}^{\infty} \phi(N) dN d\mathbf{N}^{n-1} \\ &= \int_{\mathbf{N}^{n-1} \in \mathbb{R}^{n-1}} -p(\mathbf{N}^{n-1}) \log p(\mathbf{N}^{n-1}) d\mathbf{N}^{n-1} \\ &= h(\mathbf{z}^{n-1}|\mathbf{x}^{n-1}) \quad . \end{aligned} \tag{7.13}$$

since $\int_{-\infty}^{\infty} \phi(N) dN = 1$. We can find H_2 similarly as

$$\begin{aligned} H_2 &= \int_{-\infty}^{\infty} -\phi(N) \log \phi(N) \int_{\mathbf{N}^{n-1} \in \mathbb{R}^{n-1}} p(\mathbf{N}^{n-1}) d\mathbf{N}^{n-1} dN \\ &= \int_{-\infty}^{\infty} -\phi(N) \log \phi(N) dN \\ &= h(\mathbf{z}^1|\mathbf{x}^1) \quad . \end{aligned} \tag{7.14}$$

Hence, by induction,

$$h(\mathbf{z}^n|\mathbf{x}^n) = nh(\mathbf{z}^1|\mathbf{x}^1) \tag{7.15}$$

which means the conditional entropy per unit time has the simple form

$$h(z|x) = h(\mathbf{z}^1|\mathbf{x}^1) = \frac{1 + \log 2\pi\sigma^2}{2} \quad . \tag{7.16}$$

7.1.2. Estimating $h(z)$

The key part of the work of Arnold and Loeliger [24], their method for estimating $h(z)$, proceeds as follows. Given that the output of our PR channel can be characterized as a hidden-Markov process, the Shannon-McMillan-Breimann theorem [25] shows that

$$-\frac{1}{n}\log(p(\mathbf{z^n})) \to h(z) \tag{7.17}$$

with probability one. But, for a given vector $\mathbf{z^n}$, estimating $p(\mathbf{z^n})$ can be done readily; as we recall from Chapter 2, the BCJR algorithm in its internal workings computes the $m_i(t)$ values (see (2.7)) which are estimates of the log of the probability of being in state i and having received the sequence $z(0), \ldots, z(t-1)$.

Conceptually, the Arnold-Loeliger process for estimating $h(z)$ is quite simple: for some word length n, just generate a number of random codewords $\mathbf{x^n}$ and their corresponding channel outputs $\mathbf{z^n}$, feed each $\mathbf{z^n}$ into the BCJR algorithm, compute $p(\mathbf{z^n})$ from the resulting final $m_i(n)$ values, and then compute $h(z)$ values and average over all sample codewords. But we have to be very careful. As we mentioned in Chapter 2, in the standard BCJR algorithm, we often delete extra constant terms from the $p(j, i, z, L)$ expressions (2.8) since the rest of the BCJR algorithm only uses differences between the $m_i(t)$ values, and we were allowed to renormalize the $m_i(t)$ values at will. But here we need the actual, unmodified, values of the $m_i(t)$; no discarding of seemingly-extraneous constant terms is permissible, and we must be careful about renormalizing. This imposes a bit of extra work, but on the other hand, we need only the $m_i(t)$ for our capacity calculations, not the $\bar{m}_i(t)$ or the final $L(t)$ values. Hence, we only need to do part of the BCJR algorithm for our purposes. We now explain, in detail, how to do this modified BCJR for estimating $h(z)$.

1. Generate a random input channel word $\mathbf{x^n}$.

2. Compute the channel output word $\mathbf{z}^n$. (We assume that the preceding channel inputs $x(-1), \ldots, x(-L)$ are known and fixed, and are whatever values are needed to force our PR channel to be in the known start state s at time $t = 0$).

3. Initialize $m_i(0)$ as follows:

$$m_s(0) = 0 \quad \text{and} \quad m_i(0) = -\infty \quad \forall i \neq s \qquad . \tag{7.18}$$

4. For all $t \in [1, n]$ successively compute

$$m_i(t) = \log \sum_{j:(j,i)\in D} \exp\left(m_j(t-1) + p(j, i, z(t-1), 0)\right) \quad \forall i \tag{7.19}$$

where

$$p(j, i, z, 0) = -\frac{(z - O(j,i))^2}{2\sigma^2} - \frac{\log 2\pi\sigma^2}{2} - \log(2) \tag{7.20}$$

(note that the prior likelihoods $L_{\text{in}}(t)$ are all zero, since we assume equally likely inputs $x(t)$). Since the $m_i(t)$ may get large and negative enough to cause problems with the log-sum-exponential functions (being so large that $\exp(m_i(t)) \approx 0$ in floating-point arithmetic), we may need to renormalize the $m_i(t)$ by adding a constant $\lambda(t)$ to all of them, but we must keep track of these renormalizing constants $\lambda(t)$ so we can subtract them off again at the end.

5. Compute

$$\log p(\mathbf{z}^n) = \left(\log \sum_{i=0}^{M-1} \exp(m_i(n))\right) - \sum_{t=1}^{n} \lambda(t) \qquad . \tag{7.21}$$

This is just the log of the sum of the probabilities over all final states at time n, subtracting off the constants $\lambda(t)$ that we had previously added during renormalization.

6. Compute

$$h(z) = -\frac{p(\mathbf{z}^n)}{n} \quad . \tag{7.22}$$

7. Repeat Steps 1 through 6 for as many codewords as desired and average the resulting $h(z)$ values to get a final estimate of $h(z)$.

Once we have $h(z)$, we can compute the channel capacity (in nats per unit time)

$$C_{\text{nat}}(\sigma^2) = h(z) - h(z|x) = h(z) - \frac{1 + \log 2\pi\sigma^2}{2} \quad . \tag{7.23}$$

The capacity in bits per unit time is just

$$C_{\text{bit}}(\sigma^2) = \frac{C_{\text{nat}}(\sigma^2)}{\log 2} \quad . \tag{7.24}$$

7.2. Channel Capacity Bounds on Bit Error Rate

We now know how to compute the channel capacity, in bits per unit time, for any given PR channel at any given noise level σ^2. However, as we mentioned earlier, we would like to be able to compute a bound for the best possible bit error rate for a given noise level, given the constraints of the channel having a limited capacity. We can convert our channel capacity results to BER versus SNR curves, however. To do so, we first need to know the code rate of the code whose BER performance we wish to bound; different code rates will result in different bounding curves. This is not unexpected; intuitively, it is clear that for lower code rates we should be able to get better (lower) BER for any given noise level.

Given the code rate R and the channel capacity at a given noise level $C_{\text{bit}}(\sigma^2)$, the Shannon rate equivalence theorem [11] tells us that the best achievable BER p_E

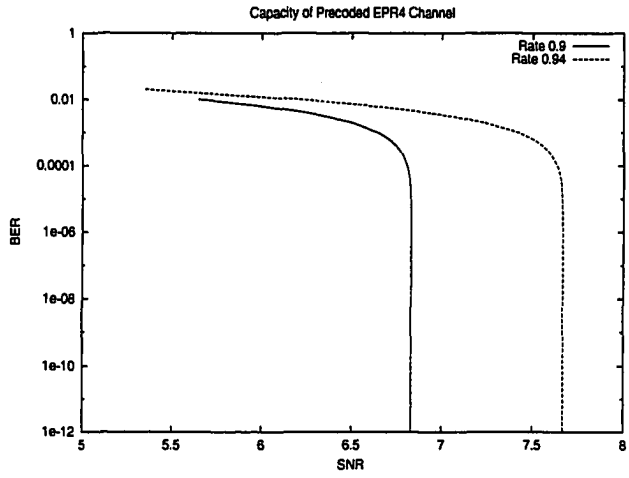


Figure 7.1: Channel capacity bound for EPR4 with $1/(1 \oplus D^2)$ precoder.

satisfies

$$R(1 - h(p_E)) = C_{\text{bit}}(\sigma^2) \tag{7.25}$$

where $h(p_E)$ is the binary symmetric channel capacity (in bits)

$$h(p_E) = -p_E \log_2 p_E - (1 - p_E) \log_2 (1 - p_E) \quad . \tag{7.26}$$

Note that if the SNR is high enough for $C_{\text{bit}}(\sigma^2) > R$, there is no solution for the above equation; asymptotically error-free transmission is possible at such SNRs. To create BER versus SNR curves for any given code rate, all we must do is compute $C_{\text{bit}}(\sigma^2)$ for various SNRs and for each SNR, solve (7.25) for p_E. Since the computation for each SNR is somewhat lengthy and the desired BER curves often vary sharply over small variations in SNR, we often find it useful to compute $C_{\text{bit}}(\sigma^2)$ for a small number of SNR values and linearly interpolate them before computing the final BER curves. Fig. 7.1 shows the capacity bounds for the EPR4 precoded channel at code rates 0.9 and 0.94. Fig. 7.2 shows the same curves for the case of the precoded MEEPR4 channel.

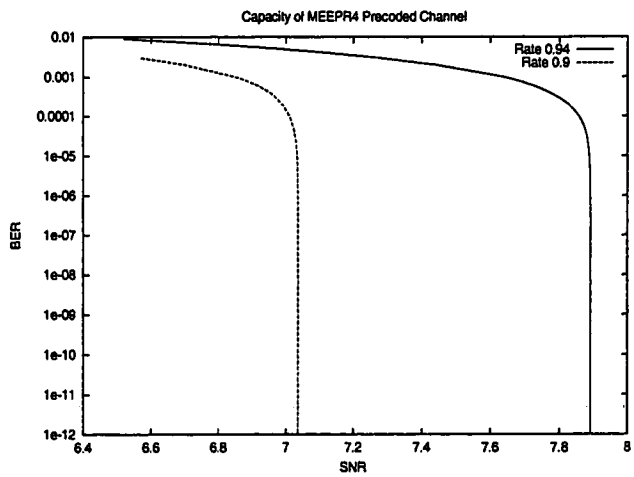


Figure 7.2: Channel capacity bound for MEEPR4 with $1/(1 \oplus D)$ precoder.

Chapter 8

LDPC Code Design for the PR Channel and BCJR Density Evolution

In this chapter we discuss how to design LDPC codes for use with PR channels, that is to say, finding sets of code parameters λ_i, ρ_i which will lead to codes that perform well over such channels. To do this, we need to be able to analyze the performance of LDPC codes with given code parameters when used with PR channels. In the case of LDPC codes used with memoryless channels (e.g. AWGN), the density evolution techniques of Chapter 5 allow us to do such analysis. But in order to be able to do density evolution for the PR channel case, we need some way of computing the probability density function of the output of the BCJR decoder. In effect, we need to be able to do density evolution to analyze the behavior of the BCJR decoder. Below we explain how to carry out density evolution for the BCJR decoder and then how to use the BCJR and LDPC density evolution algorithms to search for good λ_i, ρ_i sets.

8.1. BCJR Density Evolution

In this section we will use the formulation of the BCJR algorithm given in Chapter 2. We will also use the notation and methods for representing probability density functions (pdfs) introduced in Section 5.1. We shall also assume that there is no turbo equalization in use, i.e., that we do not feed likelihoods back from the LDPC decoder to the BCJR decoder. Extending the subsequent analysis of the BCJR to allow for turbo equalization would not be overly difficult, but, as we shall discuss later in the section on searching for good codes, the search becomes impractically computationally expensive unless we assume no turbo equalization.

One of the things that makes doing density evolution of the BCJR algorithm difficult is that, unlike the LDPC algorithm where we could assume that the input codeword was all-zeros without loss of generality, the BCJR algorithm's behavior is non-trivially dependent on the channel inputs $x(t)$. Hence what we have to do

is do density evolution for the BCJR decoder with several different possible input sequences $x(t)$ and then combine the results to get an effective pdf that we can use as input to the LDPC density evolver. In the next subsection we show how to do BCJR density evolution for a known fixed channel input sequence $x(t)$. In the following section we will build on this result to get the pdf for the BCJR outputs $L(t)$ in the case where the $x(t)$ are not known.

8.1.1. BCJR Density Evolution: Fixed Input $x(t)$ Case

Given the known (fixed) input codeword $x(t)$, it is a simple matter to compute the ideal (no-noise) channel output $\hat{z}(t)$ for all t; we just have

$$\hat{z}(t) = \sum_{k=0}^{L} g_k x(t-k) \quad . \tag{8.1}$$

The actual channel output $z(t)$ is $\hat{z}(t)$ corrupted by AWGN, so $z(t)$ is as follows:

$$z(t) = n(t) + \hat{z}(t) \quad n(t) \sim N(0, \sigma^2) \quad . \tag{8.2}$$

For the case we are interested in, i.e., no turbo equalization, the *a priori* likelihoods L are always 0. Given that, $p(j, i, z(t), L)$ (2.8) simplifies to

$$\begin{aligned} p(j, i, z(t), 0) &= -\frac{1}{2\sigma^2}\left(n(t) + \hat{z}(t) - O(j,i)\right)^2 \\ &= -\frac{1}{2\sigma^2} n(t)^2 \\ &\quad -\frac{1}{2\sigma^2}\left(2n(t)\left(\hat{z}(t) - O(j,i)\right) + \left(\hat{z}(t) - O(j,i)\right)^2\right) \end{aligned} \quad . \tag{8.3}$$

Now, the $n(t)^2$ term does not depend on j or i, so this term appears in the equations for all the $m_i(t)$ independent of i. Since only differences between the $m_i(t)$ matter in the BCJR algorithm, we can make the $n(t)^2$ term disappear into our allowed renor-

malization of the $m_i(t)$. Hence we only need to consider the pdf of the remaining terms:

$$\begin{aligned} \hat{p}(j,i,z(t),0) &= -\frac{1}{2\sigma^2}\left(2n(t)w(t,i,j)+w(t,i,j)^2\right) \\ &= -\frac{n(t)w(t,i,j)}{\sigma^2} - \frac{w(t,i,j)^2}{2\sigma^2} \end{aligned} \tag{8.4}$$

where we have defined

$$w(t,i,j) = \hat{z}(t) - O(i,j) \tag{8.5}$$

as the distance between $\hat{z}$ (the ideal noise-free channel output) at time t and some possible channel output $O(j,i)$, and the hat on $\hat{p}(j,i,z(t),0)$ indicates we have dropped those $n(t)^2$ terms. Since $w(t,i,j)$ is not stochastic and $n(t)$ is known to be Gaussian, we know that

$$\hat{p}(j,i,z(t),0) \sim N\left(-\frac{w(t,i,j)^2}{2\sigma^2}, \frac{w(t,i,j)^2}{\sigma^2}\right) \tag{8.6}$$

We can thus use the Gaussian distribution to compute $\mathcal{P}[\hat{p}(j,i,z(t),0)]$. In practice, $w(t,i,j)$ only takes on a fixed number of values (each value corresponding to a single-step distance between two paths in the channel trellis), so given σ^2 we can easily precompute a table of all possible $\mathcal{P}[\hat{p}(j,i,z(t),0)]$ vectors.

We have the expression for the $m_i(t)$ given before:

$$m_i(t) = \max_{j:(j,i)\in D} m_j(t-1) + \hat{p}(j,i,z(t-1),0) \quad \forall i \tag{8.7}$$

(note that we are here employing the max function to approximate the log-sum-exponential). Given this and knowledge of $\mathcal{P}[m_j(t-1)]$ for all j, and given (8.6) to compute $\mathcal{P}[\hat{p}(j,i,z(t-1),0)]$, we have

$$\mathcal{P}[m_i(t)] = \widetilde{\max}_{j:(j,i)\in D} \mathcal{P}[m_j(t-1)] \tilde{+} \mathcal{P}[\hat{p}(j,i,z(t-1),0)] \tag{8.8}$$

and so we can recursively compute pdfs for all $m_i(t)$. The recursion for computing $\mathcal{P}[\bar{m}_i(t)]$ proceeds in much the same fashion, with the obvious changes for going backwards instead of forwards:

$$\mathcal{P}[\bar{m}_i(t)] = \widetilde{\max}_{j:(i,j)\in D}\mathcal{P}[\bar{m}_j(t+1)]\tilde{+}\mathcal{P}[\hat{p}(i,j,z(t),0)] \quad . \tag{8.9}$$

In both cases, the recursion is initialized with sets of pdfs derived from the initialization conditions for the BCJR (2.6), (2.15), i.e.,

$$\mathcal{P}[m_s(0)]_k = \delta_{k,0} \quad \mathcal{P}[m_i(0)]_k = \delta_{k,-K} \quad \forall i \neq s \tag{8.10}$$

where $-K$ is some suitably large negative integer representing the negative infinity with which we initialize $m_i(0)$. Similarly, we set

$$\mathcal{P}[m_e(N)]_k = \delta_{k,0} \quad \mathcal{P}[m_i(N)]_k = \delta_{k,-K} \quad . \tag{8.11}$$

The next step is to compute pdfs for $L_0(t)$, $L_1(t)$, and finally $L(t)$. Given the previously computed pdfs, we can find

$$\begin{aligned}\mathcal{P}[L_0(t)] \;=\; & \widetilde{\max}_{(i,j)\in D, I(i,j)=0}\mathcal{P}[m_i(t)]\tilde{+}\mathcal{P}[\hat{p}(i,j,z(t),0)] \\ & \tilde{+}\mathcal{P}[\bar{m}_j(t+1)]\end{aligned} \tag{8.12}$$

$$\begin{aligned}\mathcal{P}[L_1(t)] \;=\; & \widetilde{\max}_{(i,j)\in D, I(i,j)=1}\mathcal{P}[m_i(t)]\tilde{+}\mathcal{P}[\hat{p}(i,j,z(t),0)] \\ & \tilde{+}\mathcal{P}[\bar{m}_j(t+1)]\end{aligned} \tag{8.13}$$

and finally

$$\mathcal{P}[L(t)] = \mathcal{P}[L_0(t)]\tilde{-}\mathcal{P}[L_1(t)] \quad . \tag{8.14}$$

Note that this gives a different pdf for each time t; in the next section we discuss

how to combine these into an "average" pdf, and also averaging over many different input sequences to get a general $\mathcal{P}[L]$ suitable for use in density evolution of combined BCJR/LDPC systems.

8.1.2. BCJR Density Evolution with Unknown Input Codewords

In the previous section we showed how to compute the $\mathcal{P}[L(t)]$ that result from the BCJR algorithm given a specific input codeword $x(t)$. To compute a general $\mathcal{P}[L]$ usable when we do not have a given fixed input codeword, we compute the $\mathcal{P}[L(t)]$ for a set of various codewords $x_j(t)$ and average the result. We average both over different codewords and different values of t. The set of codewords we use for $x_j(t)$ are a set of 2^k codewords defined as follows:

$$x_j(t) = \begin{cases} 1 & \text{if } j \wedge 2^{t \bmod k} \neq 0 \\ 0 & \text{if } j \wedge 2^{t \bmod k} = 0 \end{cases} \tag{8.15}$$

where $\wedge$ is the Boolean and operator. This definition is written such that $x_0(t)$ is the length k sequence of bits $(0, 0, \ldots, 0)$ repeated over and over, $x_1(t)$ is the length k sequences $(1, 0, \ldots, 0)$ repeated, and the set of $x_j(t)$ covers all possible repeating sequences of k bits. As k increases, averaging the $\mathcal{P}[L(t)]$ for each sequence should give better and better approximations to the "ideal" likelihood pdf.

We have to be a bit careful about how we do the averaging, though. To see why, suppose that for some pair of codewords $x_i(t), x_j(t)$ at some given t we have $x_i(t) = 0$ and $x_j(t) = 1$. Performing the BCJR algorithm on the x_i codeword should, intuitively, give a likelihood for this bit of that codeword $L_i(t)$ tending towards the positive, with a mean $E[L_i(t)] > 0$. Performing the BCJR algorithm with channel inputs $x_j(t)$ will, correspondingly, give an $E[L_j(t)] < 0$. Averaging the

two corresponding $\mathcal{P}[L_i(t)], \mathcal{P}[L_j(t)]$ pdfs is not going to give a meaningful result, as the distribution of variations in L we are interested in due to how the noise affects the decoder will be overwhelmed by the variations resulting from combining the differing signs of the means $E[L_i(t)], E[L_j(t)]$. Furthermore, we are interested in getting a likelihood pdf we can use as input for the LDPC density evolution algorithm from Chapter 5, and that algorithm computes likelihood pdfs based on the assumption that the codeword is all-zero. So what we need to do, before combining our various pdfs by averaging them, is to compensate for these differences between the $L_i(t)$s for the one and zero bits by "flipping" the pdfs for the $L_i(t)$s for which $x(t) = 1$ before averaging. To put this more precisely, instead of dealing with $L_i(t)$ values and their pdfs, we deal with the related values

$$\tilde{L}_i(t) = L_i(t)(-1)^{x_i(t)} \tag{8.16}$$

which results in just changing the sign for $L_i(t)$ if the tth bit of the ith codeword is 1. This operation induces a corresponding operation on the pdfs of the Ls and $\tilde{L}$s:

$$\begin{aligned} \mathcal{P}\left[\tilde{L}_i(t)\right]_\ell &= \mathcal{P}\left[L_i(t)(-1)^{x_i(t)}\right]_\ell \\ &= \mathcal{P}[L_i(t)]_{\ell(-1)^{x_i(t)}} \quad . \end{aligned} \tag{8.17}$$

We then average these pdfs over all i and a range of ts to get a final pdf $\mathcal{P}[L]$ suitable for inputing into the LDPC density evolution algorithms:

$$\mathcal{P}[L]_\ell = \frac{1}{2^k} \frac{1}{t_1 - t_0 + 1} \sum_{i=0}^{2^k-1} \sum_{t=t_0}^{t_1} \mathcal{P}\left[\tilde{L}_i(t)\right]_\ell \tag{8.18}$$

where $t_0 \leq t \leq t_1$ is the time interval of interest. In practice, for efficiency reasons, we do not average over the entire codeword length $0 \leq t < N$ but instead over an interval of length k somewhere in the middle of the codeword. This is based on

the assumption that the codewords are long (thousands of bits) and that, except for regions near one end or the other, the behavior of the BCJR algorithm should be reasonably uniform in the region away from either end. Thus, in averaging over this interval, we are assuming that such edge effects are negligible. We now have a combined or effective $\mathcal{P}[L]$ pdf of the output of the BCJR decoder that we can use as input for the LDPC density evolution algorithm.

8.2. Searching for Good Codes

We now describe how, given the density evolution algorithm described in the previous section which can evaluate the performance of a class of LDPC codes, we search for a good class of such codes (i.e., for a good set of λ_i, ρ_i values). Our algorithm here is essentially the one in [26]. First, one picks a set of subscript bounds $d_{lmin}, d_{lmax}, d_{rmin}, d_{rmax}$ which specify the range of subscripts for which λ_i, ρ_i are allowed to be nonzero; more precisely:

$$\begin{aligned} \lambda_i = 0 &\iff i \notin [d_{lmin} - 1, d_{lmax}] \\ \rho_i = 0 &\iff i \notin [d_{rmin} - 1, d_{rmax}] \end{aligned} \quad . \tag{8.19}$$

We specify a set of λ_i, ρ_i values with a vector

$$\mathbf{v} = (\lambda_{d_{lmin}}, \ldots, \lambda_{d_{lmax}-1}, \rho_{d_{lmin}}, \rho_{d_{lmax}}) \quad . \tag{8.20}$$

These values are enough to completely specify all our λ_i, ρ_i, since the λ_i, ρ_i must satisfy the following constraints:

$$\begin{aligned} \sum_i \lambda_i &= 1 \\ \sum_i \rho_i &= 1 \\ \sum_i \frac{\rho_i}{i} &= (1-R)\sum_i \frac{\lambda_i}{i} \end{aligned} \tag{8.21}$$

where R is the code rate. Hence from the values given in our $\mathbf{v}$ we can compute the other needed λ_i, ρ_i values as follows:

$$\lambda_{d_{lmax}} = \frac{\left(S_\rho + \frac{1}{d_{rmin}-1} - (1-R)S_\lambda - \frac{1-R}{d_{lmin}-1}\right)}{(1-R)\left(\frac{1}{d_{lmax}} - \frac{1}{d_{lmin}-1}\right)} \tag{8.22}$$

where

$$S_\rho = \sum_{i=d_{rmin}}^{d_{rmax}} \rho_i \left(\frac{1}{i} - \frac{1}{d_{rmin}-1}\right) \tag{8.23}$$

and

$$S_\lambda = \sum_{i=d_{lmin}}^{d_{lmax}-1} \lambda_i \left(\frac{1}{i} - \frac{1}{d_{lmin}-1}\right) \tag{8.24}$$

and then we can use the constraints that the λ_i, ρ_i sum to one to solve for the remaining values $\lambda_{d_{lmin}-1}, \rho_{d_{rmin}-1}$. ([26] presents the above equations in the restricted case that $d_{lmin} = d_{rmin} = 3$.) Note that not all vectors $\mathbf{v} \in [0,1]^n$ lead to acceptable sets of λ_i, ρ_i. Some $\mathbf{v}$ choices may, when (8.22) and the other equations are solved, lead to $\lambda_{d_{lmax}}, \lambda_{d_{lmin}-1}$ or $\rho_{d_{rmin}-1}$ values that are out-of-bounds (outside the interval $[0,1]$). Such $\mathbf{v}$ vectors are called *non-admissible* and those which do lead to valid λ_i, ρ_i sets are called *admissible*.

We now describe the algorithm for searching for good LDPC code parameter sets λ_i, ρ_i:

1. Select an initial noise variance σ^2 and an increment $\delta\sigma^2$. Also select d_{lmin}, d_{rmin}, d_{lmax}, d_{rmax}.

2. Generate a set of M random admissible vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{M-1}$.

3. For each vector $\mathbf{v}_i$ evaluate the corresponding set-of-codes' performance via density evolution, getting a value $P(\mathbf{v}_i)$.

4. Let i_{min} be the index of the $P(\mathbf{v}_i)$ which gives the minimum value. This corresponds to our provisional best vector $\mathbf{v}_{i_{min}}$.

5. If all the $P(\mathbf{v}_i)$ are "too large" (we define this as $P(\mathbf{v}_i) > 0.001$), we are at too high a noise variance. In this case, we do

$$\begin{aligned} \sigma^2 &\leftarrow \sigma^2 - \delta\sigma^2 \\ \delta\sigma^2 &\leftarrow \frac{\delta\sigma^2}{2} \end{aligned}$$

   and go back to Step 3.

6. For each $i = 0, \ldots, M-1$, we create a new vector

$$\mathbf{v}_{new} = \mathbf{v}_i + 0.5(\mathbf{v}_a - \mathbf{v}_b + \mathbf{v}_c - \mathbf{v}_d) \tag{8.25}$$

   where a, b, c, d are random numbers in $[0, M-1]$. If $\mathbf{v}_{new}$ is not admissible, keep trying new a, b, c, d 4-tuples until we get an admissible one. Then compute $P(\mathbf{v}_{new})$; if the resulting probability is smaller than $P(\mathbf{v}_i)$, replace $\mathbf{v}_i$ with $\mathbf{v}_{new}$.

7. Add $\delta\sigma^2$ to σ^2 and go back to Step 3.

The algorithm has no explicit termination condition; in practice, we let the algorithm run until it seems to make no further progress, usually in a state where one of the $P(\mathbf{v}_i)$ is reasonably small, all the other $P(\mathbf{v}_i)$ are large, and $\delta\sigma^2$ is small (10^{-4} or so). We then take the values from $v_{i_{min}}$ to design our good LDPC code.

Table 8.1: λ_i, ρ_i values for LDPC code designed for precoded EPR4

i	λ_i
3	0.2131352
6	0.0024832
7	0.3705876
8	0.0000540
10	0.4137401

i	ρ_i
59	0.3731251
60	0.4642554
63	0.0003370
64	0.1499316
65	0.0014538
69	0.0000522
72	0.0000039
73	0.0002034
75	0.0008222
76	0.0000163
78	0.0000009
80	0.0000839
86	0.0097144

8.3. Code Design Example and Simulation Results

Here we present an example of a code designed using the technique we described previously. The target channel is EPR4 ($h(D) = 1 + D - D^2 - D^3$) with a $\frac{1}{1 \oplus D^2}$ precoder. We chose a code rate $R = 4352/4835 \approx 0.90$ and weight bounds $d_{lmin} = 4$, $d_{lmax} = 10$, $d_{rmin} = 60$, and $d_{rmax} = 90$. The resulting λ_i, ρ_i are in Table 8.1.

To explore the performance of codes with this set of λ_i, ρ_i code parameters, we generated a set of four LDPC codes of various block lengths. The codes are:

- Code 1: a regular column weight three code with $K = 4352, N = 4835$. This is the code from [9].
- Code 2: a code based on our computed λ_i, ρ_i with $K = 4352, N = 4385$.
- Code 3: a regular column weight three code four times as big as Code 1, i.e., $K = 17408, N = 19340$.
- Code 4: a code based on our computed λ_i,ρ_i with $K = 17408, N = 19340$.

Results from simulations of the shorter ($N = 4835$) codes are given in Fig. 8.1. The parameters of the simulation are as follows: we used 5 iterations of turbo equalization and an iteration limit of 30 on the LDPC decoder. The definition of SNR used here is SNR $= 4/\sigma^2$, where 4 is the total power of the EPR4 impulse response and σ^2 is the EPR4 channel noise variance. The plots also show the capacity bound for codes of rate 0.9, as computed with the Arnold-Loeliger method from the previous chapter. As we can see, both codes are short a bit over 2 dB from achieving the capacity bound, and, unfortunately, our new code performs 0.2dB worse that our already-existing Code 1. We hypothesized that this deviation was due to the density-evolution technique only theoretically working in the limit of $N \to \infty$ and, hence, not necessarily being an accurate predictor of performance at smaller N. That hypothesis is why we decided to create the longer Codes 3 and 4 mentioned above and test their performance. Unfortunately, the situation in this case, as shown in Fig. 8.2, is even worse, as our supposedly-optimized Code 4 performs about 1dB worse than the regular Code 3. Clearly, these are not the sort of results we were hoping for. Our current theory is that our choice of weight bounds $d_{lmin} = 4$, $d_{lmax} = 10$ was inauspicious, as it provides the code with a great many columns of high weight, which causes the code to have a great many cycles. As we said before, cycles impair the performance of LDPC codes, and the effects of cycles are not modeled by the density evolution technique. We suspect it would be better for d_{lmax} to be at most six, or possibly even lower still.

We attempted to confirm this hypothesis by creating two more codes with lower ranges for the column weights, namely $d_{lmin} = 4$, $d_{lmax} = 5$, $d_{rmin} = 30$, and $d_{rmax} = 90$. The resulting λ_i, ρ_i are shown in Table 8.2. As before, we created codes of length 4835 (Code 5) and length 17408 (Code 6). Performance of those codes compared to the regular weight 3 codes is shown in Figs. 8.3 and 8.4. The graphs show that Code 5 provides a gain of approximately 0.5dB over the regular weight

three code, Code 1. Unfortunately, this is not the case with the larger block length codes, Code 6 versus Code 3; here we get approximately a 0.75dB loss, which is somewhat unexpected given the success of the smaller block length code.

We also performed a search for good code parameters for the case of precoded MEEPR4. The parameters of the search were $d_{lmin} = 4$, $d_{lmax} = 8$, $d_{rmin} = 30$, and $d_{rmax} = 80$. The resulting code specification λ_i, ρ_i is in Table 8.3. We used this specification to build a block-length 4835 code, which we call Code 7. Fig. 8.5 gives the results of a simulation of this code and our regular column weight Code 1 over a Lorentzian channel equalized to MEEPR4. As before, the number of turbo equalization and LDPC iterations were 5 and 30, respectively; the channel is a Lorentzian channel with density $S = 3.3$, and the LDPC codes are used in combination with the MTR code [2] as discussed in Chapter 1, as well as a $1/(1 \oplus D)$ precoder. The signal-to-noise ratio is defined as $SNR = 1/\sigma^2$ where σ^2 is the variance of the noise at the Lorentzian channel output (i.e., **not** the noise at the output of the MEEPR4 equalizer). We also give a capacity bound curve; this is not the true capacity bound for the MEEPR4-equalized Lorentzian channel, since there is no known method to calculate that. Instead, it is just the capacity for a perfect MEEPR4 channel computed as in Chapter 7, but with the SNR axis rescaled to bring it in line with the different definition of SNR used above. It is expected that the true MEEPR4-equalized Lorentzian channel capacity is somewhere close to the ideal MEEPR4 curve, but as this is not known for sure, the curve should be treated as an inexact measure of where the true capacity bound lies. The plot in Fig. 8.5 show that the new code provides a gain of about 0.3dB over the regular column weight three code.

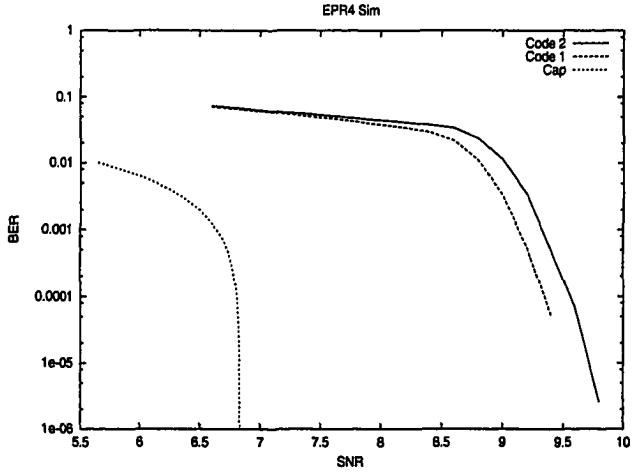


Figure 8.1: BER simulation results for block size 4835 codes over EPR4 channel.

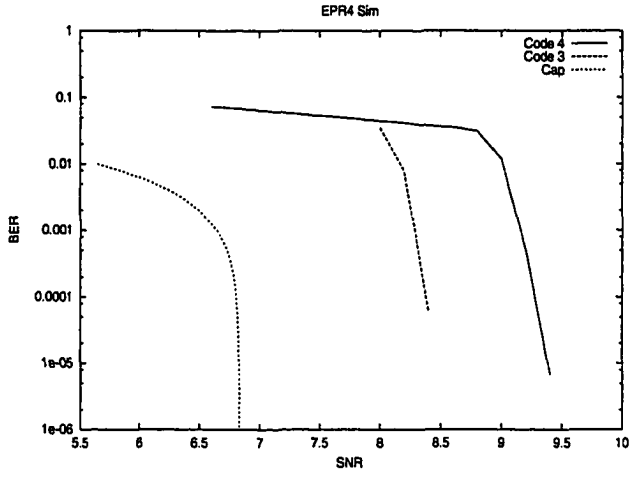


Figure 8.2: BER simulation results for block size 19340 codes over EPR4 channel.

Table 8.2: λ_i, ρ_i values for another LDPC code designed for precoded EPR4

i	λ_i
3	0.7656858
5	0.2343141

i	ρ_i
29	0.000000428210
31	0.000002141048
33	0.974863240562
34	0.000291610732
35	0.000000428210
37	0.000544254391
38	0.000012846288
39	0.022460449492
46	0.000785764600
47	0.000009420611
51	0.000829870188
54	0.000017984803
56	0.000000428210
59	0.000179419819
60	0.000001712838

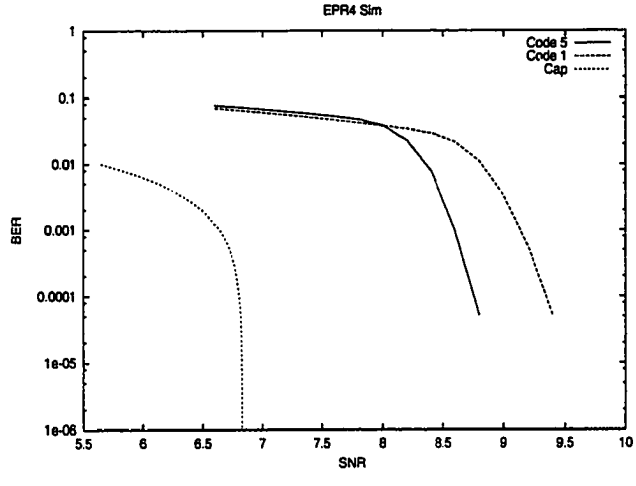


Figure 8.3: BER simulation results for block size 4835 codes over EPR4 channel.

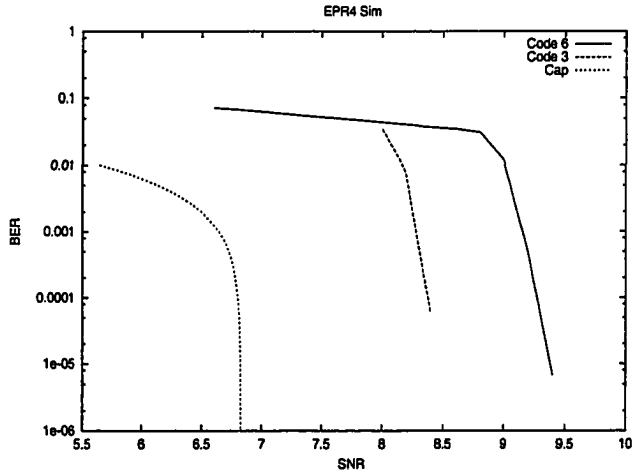


Figure 8.4: BER simulation results for block size 19340 codes over EPR4 channel.

Table 8.3: λ_i, ρ_i values for an LDPC code designed for precoded MEEPR4

i	λ_i
3	0.698047250
8	0.301952750

i	ρ_i
29	0.000000428
33	0.000002997
37	0.999126881
74	0.000000856
77	0.000002997
78	0.000864127
79	0.000001713

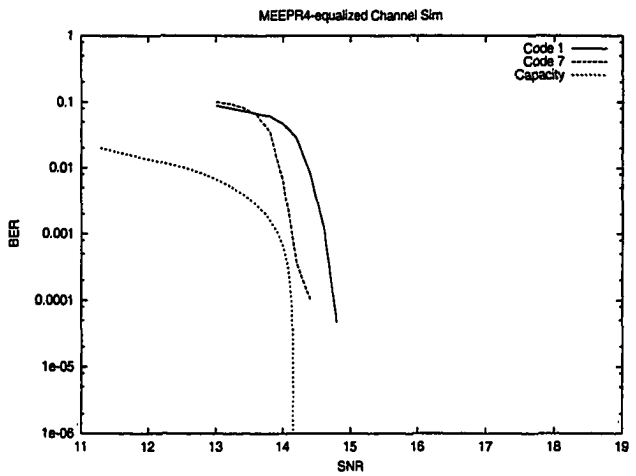


Figure 8.5: BER simulation results for block size 4352 codes over MEEPR4-equalized Lorentzian channel.

Chapter 9

Generalized Belief Propagation and Decoding of LDPC Codes

9.1. Introduction

Within the past decade, there has been a great deal of research involving two major types of error correcting codes, the turbo codes of Berrou et al. [14] and the LDPC codes of Gallager and MacKay [5], [6]. The decoding algorithms for these two types of error-correcting codes are closely related, and in fact both these algorithms have been shown [27] to be special cases of the so-called "belief propagation" (BP) algorithm developed by Pearl for general Bayesian probabilistic inference upon a graph [28]. As a result, Pearl's BP algorithm and its properties are of particular interest. Recently Yedidia et al. [29] have shown a relationship between the belief propagation algorithm and a quantity from statistical mechanics called the Bethe free energy [30]. They show that fixed points of the BP algorithm are extrema of the Bethe free energy, and go on to derive new variant BP algorithms which extremize a generalization of the Bethe free energy called the Kikuchi free energy [31]. In this chapter, we examine the work of Yedidia et al. and consider its relevance to the field of LDPC decoding.

9.2. Belief Propagation and Bethe Free Energy

First, a brief comment about the Yedidia et al. paper. The notation used and presentation of the belief propagation algorithm in [29] is somewhat different from the way belief propagation is usually described in the LDPC decoding literature and the way we described LDPC decoding in Chapter 3. The relationship between what Yedidia et al. do and the "standard" LDPC decoding algorithm may be somewhat obscure to the reader. In fact, it was obscure to one particular reader, D. J. C. MacKay. He asked for clarification on various parts of the paper that were unclear to him, and the result was a companion technical report [32] by MacKay and the original paper's authors that contains MacKay's questions and answers to those

questions. We have found this technical report useful in understanding the original Yedidia paper and recommend it be read in conjunction with said paper.

We now proceed to describe the belief propagation algorithm as presented by Yedidia et al. and show how belief propagation is related to the Bethe free energy. We start with a network of nodes, numbered 1 through N. Each of the nodes can be in one of several possible states; we use the variable x_i to describe the state of node i. (Note that the set of states node i can be in does **not** have to be the same as the set of states node j can be in. This is different from the standard description of LDPC coding as in, say, Chapter 3, where the variable and check nodes all have two states and, thus, the associated messages are two-element vectors $[q_{ji}^0, q_{ji}^1]$ or $[r_{ji}^0, r_{ji}^1]$. Later we will discuss how this formulation of BP decoding in [29] is in fact equivalent to the more traditional formulation of the LDPC decoder.) Various pairs of the nodes in our network are linked together; these links between pairs of nodes impose correlations between the states of each node. For each link in the network we are given a function $\psi_{ij}(x_i, x_j)$ which quantifies the correlation. Such a network is called a Markov Random Field. Note that we can consider each link as a link from i to j or from j to i; this implies the symmetry condition

$$\psi_{ij}(x_i, x_j) = \psi_{ji}(x_j, x_i) \quad . \tag{9.1}$$

The states x_i themselves are not observable, but we can observe other variables y_i which are related to the x_i such that by observing the y_i we can compute known *a priori* probabilities $\psi_i(x_i)$ of each node i being in a state x_i. (The $\psi_i(x_i)$ values are sometimes called the "evidence" for node i.) The probability density function for $x_1, \ldots, x_N$ given the observed y vector is

$$P(x_1, \ldots, x_N|y) = \frac{1}{Z}\left(\prod_{<ij>}\psi_{ij}(x_i, x_j)\right)\left(\prod_k \psi_k(x_k)\right) \tag{9.2}$$

where Z is a normalizing factor. Here $< ij >$ means we take the product over all **ordered** pairs (i, j) which are linked in the network and where $i < j$; i.e., if there is a link between nodes 1 and 2, we do not include both $\psi_{12}(x_1, x_2)$ and $\psi_{21}(x_1, x_2)$ in the product, but just include one (the former, though by (9.1) it does not matter which one of the two we include, as long as we only include one of them).

Anyway, the goal of the BP algorithm is to attempt to compute the *a posteriori* probabilities that each node i is in some state x_i, given the initial *a priori* probabilities $\psi_i(x_i)$ and the known correlations between nodes $\psi_{ij}(x_i, x_j)$. To do this, the BP algorithm computes messages $m_{ij}(x_j)$, which can be thought of as data propagating from node i to its neighbor j giving it some information of what the probability should be of node j being in state x_j, given the information known at node i. The BP algorithm also computes so-called "beliefs" $b_i(x_i)$, which eventually (one hopes) converge to the desired *a posteriori* probabilities $P(x_i|y)$. The messages $m_{ij}(x_j)$ start out initialized to all ones, implying no knowledge giving one a reason to pick one state over another. One then iterates between the following two rules for updating the beliefs and messages:

$$m_{ij}(x_j) \leftarrow \alpha \left(\sum_{x_i} \psi_{ij}(x_i, x_j) \psi_i(x_i) \prod_{k \in N(i) \backslash j} m_{ki}(x_i) \right) \tag{9.3}$$

and

$$b_i(x_i) \leftarrow \alpha \left(\psi_i(x_i) \prod_{k \in N(i)} m_{ki}(x_i) \right) \tag{9.4}$$

where $N(i)$ is the set of all nodes that are connected to (i.e., neighbors of) node i and $N(i) \backslash j$ is the same as $N(i)$ but with node j removed. As in Chapter 3, $\alpha()$ is a normalization operator; for each i, j pair one computes $m_{ij}(x_j)$ for all possible states x_j node j can be in, and then divides all these $m_{ij}(x_j)$ by some scale factor

such that

$$\sum_{x_j} m_{ij}(x_j) = 1 \tag{9.5}$$

and similarly each set of $b_i(x_i)$ values is renormalized such that

$$\sum_{x_i} b_i(x_i) = 1 \quad . \tag{9.6}$$

One can also, at each stage of the iteration, compute pairwise belief functions for each pair of connected nodes in the network

$$b_{ij}(x_i, x_j) = \alpha \left(\phi_{ij}(x_i, x_j) \prod_{k \in N(i) \backslash j} m_{ki}(x_i) \prod_{\ell \in N(j) \backslash i} m_{\ell j}(x_j) \right) \tag{9.7}$$

where the functions ϕ_{ij} are defined as follows:

$$\phi_{ij}(x_i, x_j) \equiv \psi_i(x_i) \psi_{ij}(x_i, x_j) \psi_j(x_j) \quad . \tag{9.8}$$

If the network is a tree, it is known [28] that the $b_i(x_i)$ converge to the exact marginal probabilities $P(x_i|y)$ and the $b_{ij}(x_i, x_j)$ converge to the exact marginal probabilities $P(x_i, x_j|y)$. For networks with loops, things are more complicated, but one can derive a theorem regarding fixed points of the BP algorithm.

First, we define a quantity called the Bethe free energy [30], which is a function of the $b_{ij}(x_i, x_j)$ and $b_i(x_i)$:

$$\begin{aligned} F_\beta(b_{ij}, b_i) \;=\; & -\sum_{<ij>} \sum_{x_i, x_j} b_{ij}(x_i, x_j) \ln \phi_{ij}(x_i, x_j) \\ & + \sum_i (q_i - 1) \sum_{x_i} b_i(x_i) \ln \psi_i(x_i) \\ & + \sum_{<ij>} \sum_{x_i, x_j} b_{ij}(x_i, x_j) \ln b_{ij}(x_i, x_j) \\ & - \sum_i (q_i - 1) \sum_{x_i} b_i(x_i) \ln b_i(x_i) \end{aligned} \tag{9.9}$$

where q_i is the number of neighbors node i has. We can rearrange the above expression a bit, given the definition of ϕ_{ij} and the constraints that the $b_i(x_i)$ and $b_{ij}(x_i, x_j)$ must satisfy:

$$\begin{aligned}
\sum_{x_i,x_j} b_{ij}(x_i,x_j) &= 1 \\
\sum_{x_i} b_i(x_i) &= 1 \\
\sum_{x_i} b_{ij}(x_i,x_j) &= b_j(x_j) \\
\sum_{x_j} b_{ij}(x_i,x_j) &= b_i(x_i) \qquad .
\end{aligned} \tag{9.10}$$

Note that the first term of $F_\beta(b_{ij}, b_i)$ can be rewritten as

$$\begin{aligned}
F_{\beta_1} &= -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) - \sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_i(x_i) \\
&\quad -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_j(x_j) \\
&= -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) - \sum_{ij}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_i(x_i)
\end{aligned} \tag{9.11}$$

(note the second sum is over unordered pairs i, j, i.e., if there is a link between nodes i and j, both pairs i, j and j, i appear in the sum) and hence

$$\begin{aligned}
F_{\beta_1} &= -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) - \sum_{ij}\sum_{x_i} b_i(x_i)\ln\psi_i(x_i) \\
&= -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) \\
&\quad -q_i\sum_{i}\sum_{x_i} b_i(x_i)\ln\psi_i(x_i)
\end{aligned} \tag{9.12}$$

and thus

$$\begin{aligned} F_\beta(b_{ij}, b_i) &= -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) - \sum_i\sum_{x_i} b_i(x_i)\ln\psi_i(x_i) \\ &\quad + \sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln b_{ij}(x_i,x_j) \\ &\quad - \sum_i (q_i-1)\sum_{x_i} b_i(x_i)\ln b_i(x_i) \end{aligned} \tag{9.13}$$

Why is this quantity called a free energy? In statistical mechanics, if a system can be described by a state vector x and we have a function $b(x)$ that may serve as a possible probability function, one defines the free energy as follows:

$$F(b(x)) = \sum_x b(x)E(x) + \sum_x b(x)\ln b(x) \tag{9.14}$$

where $E(x)$ is the energy of state x, so the free energy is just the average energy minus the entropy. If we consider our network of nodes, letting $x = (x_1, \ldots, x_N)$, and define the energy to be something consistent with Boltzmann's Law

$$P(x|y) = \frac{1}{Z}\exp(-E(x)) \tag{9.15}$$

then

$$E(x) = -\ln P(x|y) - \ln Z = -\sum_{<ij>}\ln\psi_{ij}(x_i,x_j) - \sum_i \ln\psi_i(x_i) \tag{9.16}$$

and hence the average energy is

$$\sum_x b(x)E(x) = -\sum_{<ij>}\sum_{x_i,x_j} b_{ij}(x_i,x_j)\ln\psi_{ij}(x_i,x_j) - \sum_i\sum_{x_i} b_i(x_i)\ln\psi_i(x_i) \tag{9.17}$$

where b_{ij} and b_i are the obvious marginalizations of the full $b(x)$ function down to single states or pairs of states. Thus the average energy is the first half of the Bethe

free energy given in (9.13), so we already see part of the connection between the "true" free energy and the Bethe free energy. Let us also note here that since the free energy is

$$F(b(x)) = -\ln Z - \sum_x b(x) \ln P(x|y) + \sum_x b(x) \ln b(x) \tag{9.18}$$

the free energy is obviously minimized when $b(x) = P(x|y)$, i.e., when our "trial" probability function $b(x)$ equals the true probability. Note that this free energy is a thinly disguised version of the Kullback-Liebler divergence between the two probability functions $b(x)$ and $P(x|y)$.

To see the rest of the connection, we note that, as shown in [28], if our network is known to be a tree, any $b(x)$ that is consistent with the presence or absence of correlations between the various nodes must have the form

$$b(x) = \frac{\prod_{<ij>} b_{ij}(x_i, x_j)}{\prod_i b_i(x_i)^{q_i - 1}} \tag{9.19}$$

and substituting this $b(x)$ in the expression for the entropy term of (9.14) gives us the second half of (9.13). Thus in the case of a treelike network, the Bethe free energy is **exactly** equal to the true free energy and, as such, is minimized when $b(x)$ equals the true *a posteriori* probabilities $P(x|y)$. The Bethe free energy is thus, in some sense, an approximation to the true free energy, an approximation that is exact for treelike networks. We now go on to show that, whether the network is a tree or not, the BP algorithm finds extrema of the Bethe free energy:

Theorem 1 *Suppose that we are given a network of nodes and a set of messages m_{ij} and beliefs b_{ij}, b_i. These beliefs and messages are a fixed point of the BP algorithm iff b_i and b_{ij} minimize the Bethe free energy $F_\beta(b_{ij}, b_i)$.*

Proof: We take the Bethe free energy and turn it into a Lagrangian by adding

Lagrange multipliers for the various marginalization and normalization constraints (9.10):

$$\begin{aligned} L &= F_\beta(b_{ij}, b_i) + \sum_{ij}\sum_{x_j} \lambda_{ij}(x_j)\left(b_j(x_j) - \left(\sum_{x_i} b_{ij}(x_i, x_j)\right)\right) \\ &\quad + \sum_{<ij>} \gamma_{ij}\left(1 - \sum_{x_i, x_j} b_{ij}(x_i, x_j)\right) \\ &\quad + \sum_i \gamma_i \left(\sum_{x_i} b_i(x_i)\right) - \gamma_i \quad . \end{aligned} \tag{9.20}$$

We differentiate this with respect to the $b_{ij}(x_i, x_j)$ and the $b_i(x_i)$ to get the following two equations to extremize L:

$$\begin{aligned} \frac{\partial L}{\partial b_{ij}(x_i, x_j)} &= 0 \\ \rightarrow \ln b_{ij}(x_i, x_j) &= \ln \phi_{ij}(x_i, x_j) + \lambda_{ij}(x_j) + \lambda_{ji}(x_i) + \gamma_{ij} - 1 \end{aligned} \tag{9.21}$$

and

$$\begin{aligned} \frac{\partial L}{\partial b_i(x_i)} &= 0 \\ \rightarrow (q_i - 1)(1 + \ln b_i(x_i)) &= (q_i - 1)\ln \psi_i(x_i) + \left(\sum_{j \in N(i)} \lambda_{ji}(x_i)\right) \\ &\quad + \gamma_i \quad . \end{aligned} \tag{9.22}$$

Extrema of $F_\beta(b_{ij}, b_i)$ must satisfy the previous two equations as well as the normalization and marginalization constraints (9.10). (Note: The version of (9.22) found in [29] in their proof sketch on page 6 is incorrect; they left out the $(q_i - 1)$ term that multiplies $\ln \psi_i(x_i)$.) We now proceed to prove that a fixed point of BP implies an extremum of Bethe free energy and vice versa.

1. BP fixed point implies extremum:

Suppose we have a fixed point of the BP update rules, i.e., we have

$$m_{ij}(x_j) = A_{ij} \sum_{x_i} \psi_{ij}(x_i, x_j)\psi_i(x_i) \prod_{k \in N(i)\backslash j} m_{ki}(x_i) \tag{9.23}$$

and

$$b_i(x_i) = B_i \psi_i(x_i) \prod_{k \in N(i)} m_{ki}(x_i) \tag{9.24}$$

and

$$b_{ij}(x_i, x_j) = C_{ij}\phi_{ij}(x_i, x_j) \prod_{k \in N(i)\backslash j} m_{ki}(x_i) \prod_{\ell \in N(j)\backslash i} m_{\ell j}(x_j) \tag{9.25}$$

where A_{ij}, B_i, C_{ij} are normalization constants that force the sums of $m_{ij}(x_j)$, $b_i(x_i)$, and $b_{ij}(x_i, x_j)$, respectively, to be one. Now compute new variables $\lambda_{ij}(x_j)$ from the messages as follows:

$$\lambda_{ij}(x_j) = \ln \prod_{k \in N(j)\backslash i} m_{kj}(x_j) \quad . \tag{9.26}$$

Then (9.25) becomes

$$b_{ij}(x_i, x_j) = C_{ij}\phi_{ij}(x_i, x_j) \exp(\lambda_{ji}(x_i)) \exp(\lambda_{ij}(x_j)) \tag{9.27}$$

and thus

$$\ln b_{ij}(x_i, x_j) = \ln C_{ij} + \ln \phi_{ij}(x_i, x_j) + \lambda_{ji}(x_i) + \lambda_{ji}(x_i) \quad . \tag{9.28}$$

However, this is just (9.21) with $\gamma_{ij} = 1 + \ln C_{ij}$, so we have proved one of the two conditions needed for extremizing the Bethe free energy. Now consider

the equation for the $b_i(x_i)$ (9.24). We have

$$\ln b_i(x_i) = \ln B_i + \ln \psi_i(x_i) + \ln \prod_{k \in N(i)} m_{ki}(x_i) \quad . \tag{9.29}$$

Now pick any $j \in N(i)$. We can rewrite the preceding equation as

$$\begin{aligned} \ln b_i(x_i) &= \ln B_i + \ln \psi_i(x_i) + \ln m_{ji}(x_i) + \ln \prod_{k \in N(i) \backslash j} m_{ki}(x_i) \\ \ln b_i(x_i) &= \ln B_i + \ln \psi_i(x_i) + \ln m_{ji}(x_i) + \lambda_{ji}(x_i) \quad . \end{aligned} \tag{9.30}$$

Now we can write the preceding equation for any $j \in N(i)$, so let us pick some neighbor $k \in N(i)$ and sum the versions of this equation for all $j \in N(i) \backslash k$. We get

$$\begin{aligned} (q_i - 1) \ln b_i(x_i) &= (q_i - 1) \ln B_i + (q_i - 1) \ln \psi_i(x_i) \\ &\quad + \ln \prod_{j \in N(i) \backslash k} m_{ji}(x_i) + \sum_{j \in N(i) \backslash k} \lambda_{ji}(x_i) \\ &= (q_i - 1) \ln B_i + (q_i - 1) \ln \psi_i(x_i) \\ &\quad + \lambda_{ki}(x_i) + \sum_{j \in N(i) \backslash k} \lambda_{ji}(x_i) \\ &= (q_i - 1) \ln B_i + (q_i - 1) \ln \psi_i(x_i) \\ &\quad + \sum_{j \in N(i)} \lambda_{ji}(x_i) \quad . \end{aligned} \tag{9.31}$$

But if we set

$$\gamma_i = (q_i - 1) + (q_i - 1) \ln B_i \tag{9.32}$$

we get (9.22). Hence our fixed point is indeed an extremum of the Bethe free energy.

2. Extremum implies BP fixed point: Suppose we have an extremum of the Bethe free energy and, hence, a set of $b_i(x_i)$, $b_{ij}(x_i, x_j)$, $\lambda_{ij}(x_j)$, γ_{ij}, γ_i that satisfy (9.21), (9.22), and (9.10). Now let

$$m_{ij}(x_j) = \frac{b_j(x_j)}{\exp(\lambda_{ij}(x_j))\psi_j(x_j)} \quad . \tag{9.33}$$

(Again, the sketch of the proof in [29] is in error; the $\psi_j(x_j)$ term is missing from the above equation.)

Now consider the product

$$\begin{aligned} \prod_{k \in N(i) \backslash j} m_{ki}(x_i) &= \prod_{k \in N(i) \backslash j} \frac{b_i(x_i)}{\exp(\lambda_{ki}(x_i))\psi_i(x_i)} \\ &= \frac{b_i(x_i)^{q_i - 1}}{\psi_i(x_i)^{q_i - 1} \exp\left(\sum_{k \in N(i) \backslash j} \lambda_{ki}(x_i)\right)} \quad . \end{aligned} \tag{9.34}$$

Now by (9.22) we have

$$b_i(x_i)^{q_i - 1} = \psi_i(x_i)^{q_i - 1} \exp\left(\sum_{k \in N(i)} \lambda_{ki}(x_i)\right) \exp(\gamma_i) \exp(-q_i + 1) \tag{9.35}$$

so

$$\begin{aligned} \prod_{k \in N(i) \backslash j} m_{ki}(x_i) &= \exp(\gamma_i) \exp(-q_i + 1) \exp(\lambda_{ji}(x_i)) \\ &= \exp(\gamma_i) \exp(-q_i + 1) \frac{b_i(x_i)}{\psi_i(x_i) m_{ji}(x_i)} \quad . \end{aligned} \tag{9.36}$$

Now consider

$$\begin{aligned}\psi_i(x_i)\prod_{k\in N(i)} m_{ki}(x_i) &= \psi_i(x_i)\prod_{k\in N(i)}\frac{b_i(x_i)}{\exp\left(\lambda_{ki}(x_i)\right)\psi_i(x_i)} \\ &= \psi_i(x_i)\frac{b_i(x_i)b_i(x_i)^{q_i-1}}{\exp\left(\sum_{k\in N(i)}\lambda_{ki}(x_i)\right)\psi_i(x_i)^{q_i}} \\ &= b_i(x_i)\exp(\gamma_i)\exp(-q_i+1) \qquad (9.37)\end{aligned}$$

and letting

$$B_i = \frac{1}{\exp(\gamma_i)\exp(-q_i+1)} \qquad (9.38)$$

gives us

$$B_i\psi_i(x_i)\prod_{k\in N(i)} m_{ki}(x_i) = b_i(x_i) \qquad (9.39)$$

which is just the equation specifying that $b_i(x_i)$ is part of a BP fixed point (9.24). Next we look at

$$\begin{aligned}\phi_{ij}(x_i,x_j)\prod_{k\in N(i)\setminus j} m_{ki}(x_i)\prod_{\ell\in N(j)\setminus i} m_{\ell j}(x_j) &= \phi_{ij}(x_i,x_j) \\ &\quad\times\frac{\exp(\lambda_{ji}(x_i))\exp(\lambda_{ij}(x_j))}{B_iB_j} \\ &= \frac{b_{ij}(x_i,x_j)\exp(1-\gamma_{ij})}{B_iB_j} \qquad (9.40)\end{aligned}$$

and hence we have derived (9.25) with the normalizing constants being

$$C_{ij} = \frac{B_iB_j}{\exp(1-\gamma_{ij})} \qquad (9.41)$$

Now that we have derived (9.25) and (9.24), let us combine them with the

marginalization relationship

$$\sum_{x_i} b_{ij}(x_i, x_j) = b_j(x_j) \tag{9.42}$$

to get

$$\begin{aligned} \sum_{x_i} C_{ij}\phi_{ij}(x_i, x_j) \prod_{k \in N(i)\backslash j} m_{ki}(x_i) \prod_{\ell \in N(j)\backslash i} m_{\ell j}(x_j) \\ = B_j \psi_j(x_j) \prod_{k \in N(j)} m_{kj}(x_j) \end{aligned} \tag{9.43}$$

and thus

$$\begin{aligned} \sum_{x_i} C_{ij}\psi_{ij}(x_i, x_j)\psi_i(x_i) \prod_{k \in N(i)\backslash j} m_{ki}(x_i) \prod_{\ell \in N(j)\backslash i} m_{\ell j}(x_j) \\ = B_j \prod_{\ell \in N(j)} m_{\ell j}(x_j) \end{aligned} \tag{9.44}$$

which gives us

$$\sum_{x_i} C_{ij}\psi_{ij}(x_i, x_j)\psi_i(x_i) \prod_{k \in N(i)\backslash j} m_{ki}(x_i) = B_j m_{ij}(x_j) \tag{9.45}$$

which is just the fixed-point equation for the messages (9.23) where $A_{ij} = C_{ij}/B_j$.

We have now shown that fixed points of BP are extrema of the Bethe free energy, which is an approximation to the true free energy or to the distance between our belief $b(x)$ and the true *a posteriori* probability. In practice, these extrema of the Bethe free energy usually are minima. Note that, as MacKay points out [32], this does not necessarily imply that the BP algorithm always converges to such an extremum; in fact, it is sometimes the case that the BP algorithm does not converge at all. But **if** it converges, it converges to an extremum of the Bethe free energy.

9.3. Belief Propagation and LDPC decoding

Here we explain how one applies the generic formulation of belief propagation in [29] to the specific problem of decoding of LDPC codes. As we noted before, this formulation of LDPC code decoding is rather different than the more usual formulation as in [6], [5], so it is worthwhile to examine this in detail. Recall from Chapter 3 that any LDPC code is specified by a parity-check matrix of ones and zeros, $\mathbf{H}$, of dimensions N by L where N is the codeword length, $L = N - K$ is the number of parity-check bits, and K is the number of bits available for the user's data. Each valid codeword $\mathbf{x}$ satisfies

$$\mathbf{Hx} = 0 \quad . \tag{9.46}$$

Hence, each row of $\mathbf{H}$ specifies a parity constraint that certain bits of the codeword $\mathbf{x}$ must satisfy; specifically, if row α of the $\mathbf{H}$ matrix has bits set in columns $\alpha_1, \alpha_2, \ldots, \alpha_m$, then the corresponding bits of $\mathbf{x}$ must satisfy

$$x_{\alpha_1} + x_{\alpha_2} + \cdots + x_{\alpha_m} \equiv 0 \pmod 2 \tag{9.47}$$

and we have one equation like that for each row of $\mathbf{H}$. The problem of LDPC decoding is, given *a priori* probabilities $\psi_i(x_i)$ of bit x_i in the codeword being one or zero, to find the *a posteriori* probabilities of each bit being one or zero given that we know the codeword satisfies (9.46).

To rephrase this problem in terms that the BP algorithm from the previous section can handle, we have to construct a suitable network of nodes. In this case, as one might expect, our network has two kinds of nodes, N **variable nodes** each corresponding to a codeword bit and L **check nodes** corresponding the L parity-check constraints imposed by $\mathbf{H}$. For convenience we distinguish here between the two types of nodes by using Latin letters like i as indices amongst the variable nodes and Greek letters like α as indices for the check nodes. Links exist only

between variable nodes and check nodes; there are no direct links between any pair of variable nodes or any pair of check nodes. Thus each link is between a member of one class of nodes and a member of the other class; such a network is sometimes called a **bipartite graph**. Each variable node i is linked to those parity-checks that bit i participates in. Similarly, each check node α is linked to those variable nodes which participate in the αth parity-check. Each variable node is in one of two states, corresponding to the zero and one states of the codeword bit, and $\psi_i(x_i)$ are the corresponding *a priori* probabilities for that bit. The states for the check nodes are considerably more complicated, with each node α having 2^{n_α} states, where n_α is the number of codeword bits that participate in that parity-check. Each state of node α corresponds to a possible combination of the bits that are parity-checked at that node. It is helpful to think of the check node states x_α as being themselves bit vectors, with one bit for each of the codeword bits that belong to this parity-check. The *a priori* information $\psi_\alpha(x_\alpha)$ just specifies that the bits must have even parity and no other information, so

$$\psi_\alpha(x_\alpha) = \begin{cases} 0 & \text{if } x_\alpha \text{ has odd parity} \\ \frac{1}{2^{k_\alpha - 1}} & \text{if } x_\alpha \text{ has even parity} \end{cases} \tag{9.48}$$

The compatibility matrices $\psi_{i\alpha}(x_i, x_\alpha)$ impose the constraint linking the states x_i of codeword bits to the states x_α of check nodes. If $x_\alpha(i)$ is the bit in the state x_α corresponding to bit i, then

$$\psi_{i\alpha}(x_i, x_\alpha) = \delta_{x_i, x_\alpha(i)} \tag{9.49}$$

where δ is the Kronecker delta. We now have all the information we need to perform BP on our network and find the final beliefs $b(x_i)$ for the codeword bits.

This presentation of LDPC decoding is somewhat different from that in [6] and

in Chapter 3. Here the algorithm for computing messages is the same whether the messages are messages $m_{i\alpha}(x_\alpha)$ going from variable nodes to check nodes giving the check nodes information on what state they should be in, or messages $m_{\alpha i}(x_i)$ going the other way. The only difference between the check nodes and the variable nodes is that variable nodes have binary states and check nodes have more complex states. In the conventional treatment of LDPC decoding, there are messages $q_{\alpha i}^{x_i}$ from variable nodes to check nodes telling them about the state of codeword bit i and messages $r_{\alpha i}^{x_i}$ from the check nodes to the codeword bit nodes telling them about the state of codeword bit x_i. In the conventional treatment, the check nodes are not considered to have explicit states, and the equations for computing the $r_{\alpha i}^{x_i}$ are different from those for the $q_{\alpha i}^{x_i}$; the enforcement of the parity-check constraint (the equivalent of our $\psi_\alpha(x_\alpha)$) is implicit in the $r_{\alpha i}^{x_i}$ equations (3.19). Also, in the conventional treatment all messages are two-element vectors, since x_i is always a binary variable; in this version the messages to check nodes are vectors with more than two elements. Nonetheless, these two versions of LDPC decoding are in fact equivalent and give the same results [32].

9.4. Kikuchi Free Energy and Generalized Belief Propagation

The preceding sections have shown that the BP algorithm can find extrema of the Bethe free energy, which is an approximation to the true free energy. Hence the BP algorithm attempts to minimize a quantity which approximates the distance between our beliefs $b(x)$ and the true *a posteriori* probabilities $P(x|y)$. The Bethe free energy is an approximation defined on two-node connected regions of the network (the b_{ij} and ϕ_{ij} terms) and on the intersections between those regions (the b_i and ψ_i terms). Note that not all the b_i appear in the expression for $F_\beta(b_{ij}, b_i)$, only those for which

$q_i > 1$, i.e., those b_i whose nodes are in the intersection of two two-node regions (links). This definition of the Bethe free energy can be generalized to one defined on sets of regions containing more than two nodes. This generalization is called the Kikuchi free energy [31], and it leads to a generalization of the BP algorithm.

Consider for a moment the entropy term of the Bethe free energy on the network shown in Fig. 9.1. We consider our set of fundamental connected regions, the pairs of nodes, as $\{12, 23, 36, 25, 14, 45, 56\}$ and thus the entropy term is

$$\begin{aligned} H_\beta(x) &= H(x_{12}) + H(x_{23}) + H(x_{36}) + H(x_{25}) + H(x_{14}) \\ &\quad + H(x_{45}) + H(x_{56}) - H(x_1) - 2H(x_2) - H(x_3) \\ &\quad - H(x_4) - 2H(x_5) - H(x_6) \end{aligned} \tag{9.50}$$

where the individual entropy terms are defined in the obvious way on each region r:

$$H(x_r) = -\sum_{x_r} b_r(x_r) \ln b_r(x_r) \quad . \tag{9.51}$$

Note that in our computation of $H_\beta(x)$ we subtracted $H(x_1)$ only once, since it only appears in two of our fundamental regions and thus is only over-counted once by summing over all our fundamental regions, but $H(x_2)$ gets subtracted twice because it appears in three fundamental regions. Now suppose instead we had chosen as our fundamental set regions the pair of four-node regions $\{1245, 2356\}$. Then we would compute the Kikuchi entropy over these regions, their intersections, the intersections of their intersections, etc., as

$$H_K(x) = H(x_{1245}) + H(x_{2356}) - H(x_{24}) \quad . \tag{9.52}$$

We can do a similar generalization of the energy term of the free energy. Define the

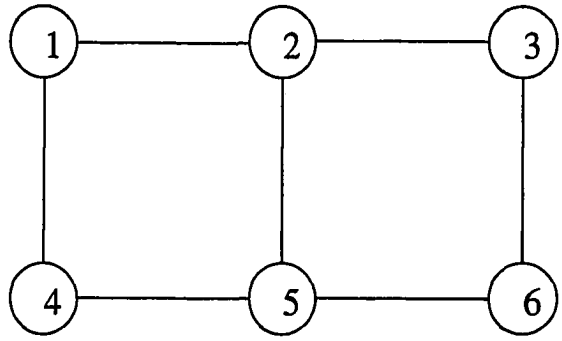


Figure 9.1: Example network for Kikuchi free energy computation.

energy associated with a region r as

$$E_r(x_r) \equiv -\ln \prod_{<ij>} \psi_{ij}(x_i, x_j) - \ln \prod_i \psi_i(x_i) \equiv -\ln \phi_r(x_r) \tag{9.53}$$

where the first product is over all links ij inside the region r and the second product is over all nodes inside r. Yedidia et al. [29] called the final term in the preceding equation $-\ln \psi_r(x_r)$, but we deviate from their notation here and believe $\phi_r(x_r)$ is a better name for these quantities. Note that for the case of a two-node region $\phi_r(x_r)$ equals the previously defined $\phi_{ij}(x_i, x_j)$ from (9.8).

We are now ready to completely define the Kikuchi free energy. Let R_0 be our set of fundamental connected regions on our network graph. Let R_1 be the set of all non-null connected intersections of regions in R_0, and let R_2 be similarly defined as the non-null connected intersections of regions in R_1, and continue on until we can continue no further. Define our total set of regions as

$$R \equiv \bigcup_i R_i \quad . \tag{9.54}$$

Note that R does not contain every possible connected subregion of our network, just those that are derivable from R_0. For example, in the network in Fig. 9.1

with the fundamental regions $\{1245, 2356\}$, the region 12 is a connected subregion of the graph, but one that is not in R. Let us write the set of all possible connected subregions of the network as $\hat{R}$. For each region $r \in R$ compute the over-counting number c_r as follows:

$$c_r = \begin{cases} 1 & \text{if } c \in R_0 \\ 1 - \sum_{s \in R, r \subset s} c_s & \text{otherwise} \end{cases} \quad . \tag{9.55}$$

c_r is thus a measure of how many times region r is over-counted by summing over all the regions that contain r. The Kikuchi free energy is now defined as

$$F_K = \sum_{r \in R} c_r \left(\sum_{x_r} b_r(x_r) E_r(x_r) + \sum_{x_r} b_r(x_r) \ln b_r(x_r) \right) \tag{9.56}$$

where the $b_r(x_r)$ are beliefs over regions $r \in R$. (Actually, we can, and will, define beliefs $b_r(x_r)$ and the corresponding $E_r(x_r)$ over any connected region r in $\hat{R}$, but only those $b_r(x_r)$, $E_r(x_r)$ for which $r \in R$ appear in the equation for the Kikuchi free energy.) This definition reduces to the Bethe free energy F_β when we let our fundamental set of regions R_0 be the set of all links in the network. The beliefs b_i, b_{ij} that the Bethe free energy depends on have to obey certain normalization and marginalization constraints. Similarly, we have constraints here on the $b_r(x_r)$; we must have

$$\begin{aligned} \sum_{x_r} b_r(x_r) &= 1 \\ \sum_{x_{r \setminus s}} b_r(x_r) &= b_s(x_s) \quad \text{if} \quad s \subset r \end{aligned} \quad , \tag{9.57}$$

i.e., each $b_r(x_r)$ must sum to one and, if we sum over all the x_i that are in r but not in the subregion s, we must get the same values as for the belief $b_s(x_s)$ over the subregion.

We now present the rules for Generalized Belief Propagation, a generalization of the BP algorithm which reduces to the standard BP algorithm in the case that R_0 is the set of all two-node links in the network, and which extremizes the Kikuchi free energy. Instead of messages from one node to another, we have messages $m_{rs}(x_s)$ from one region r to its direct subregions s. A direct subregion s of r is one such that there are no other intermediate subregions s' such that $s \subset s' \subset r$. If one represents regions r and s as bit vectors, with each bit being one if the corresponding node is in the region and zero otherwise, the direct subregion requirement can be thought of as saying that the bit vectors r and s only have Hamming distance one. (Note: Yedidia et al. [29] do not say whether the subregions s have to be **connected** subregions of the network. For the rest of this chapter, we are going to assume that s must be a connected subregion, e.g., if we have a region $1-0-2$ with no link between 1 and 2, we do not allow $s = 12$ as a possible subregion. From the limited example networks for which we have worked out explicitly what the update rules are, it appears that if one did allow messages m_{rs} into non-connected regions s, these extra messages do not affect the messages into the connected regions s, nor do they affect the single-node beliefs $b_i(x_i)$. We do not have a proof of this, however, so for now we shall just proceed with the assumption that we are restricting ourselves to connected subregions s.)

Next, we define the set $M(r)$ of messages going into the region r as follows, describing each message by its pair of regions (r, s):

$$M(r) \equiv \{(r', s') | r' \backslash s' \cap r = \{\}, s' \subseteq r\} \quad . \tag{9.58}$$

This set contains all messages whose target is inside r, but whose source contains at least one node outside r. Intuitively, it seems that the belief for a region r should depend on messages coming into r from outside r, and as we shall see momentarily,

that is indeed the case. We also need to define notation for a couple of other sets of regions. $M(r)\backslash M(s)$ is, as one might expect, the set of all messages in $M(r)$ but not in $M(s)$. $M(r,s)$ is the set of messages in $M(s)$ that originate in a proper subregion of r, i.e.,

$$M(r,s) \equiv \{(r'',s'')|(r'',s'') \in M(s), r'' \subset r\} \quad . \tag{9.59}$$

We now present the update rules for generalized BP. The messages are updated with

$$m_{rs}(x_s) \leftarrow \alpha \left(\frac{\left[\sum_{x_{r\backslash s}} \frac{\phi_r(x_r)}{\phi_s(x_s)} \prod_{(r',s') \in M(r)\backslash M(s)} m_{r's'}(x_{s'})\right]}{\prod_{(r'',s'') \in M(r,s)} m_{r''s''}(x_{s''})} \right) \tag{9.60}$$

for any $r \in R$ and s a direct subregion of r. The beliefs are updated by

$$b_r(x_r) \leftarrow \alpha \left(\phi_r(x_r) \prod_{(r',s') \in M(r)} m_{r's'}(x_{s'}) \right) \tag{9.61}$$

where r can be any subregion of $\hat{R}$. This is not quite the same set of equations as are given in [29]. Yedidia et al. used, in our notation, $\phi_{r\backslash s}(x_{r\backslash s})$ where we have $\phi_r(x_r)/\phi_s(x_s)$. We believe that Yedidia et al. are in error here, as the equations they give do not reduce to the standard BP equations when one chooses the set of two-node links as R_0. To further persuade the reader that our form of the equation is valid and leads to a generalized BP algorithm which extremizes F_K, we proceed to prove that our version of (9.60) is a straightforward consequence of (9.61) and the marginalization relations. This is our version of part of the proof presented in [29], and corresponds to our previous derivation of the message update rules from the belief update rules for the standard BP algorithm.

Suppose we have a set of messages $m_{rs}(x_s)$ at a fixed point of generalized BP and a corresponding set of beliefs $b_r(x_r)$ related via (9.61). Pick any $r \in R$ and a

direct subregion $s \subset r$. We know from the marginalization relations that

$$\sum_{x_{r\backslash s}} b_r(x_r) = b_s(x_s) \quad . \tag{9.62}$$

Since s is a direct subregion of r, $r\backslash s$ contains only one node; let us call this node c, so we have

$$\sum_{x_c} b_r(x_r) = b_s(x_s) \quad . \tag{9.63}$$

Now substitute (9.61) into this equation to get

$$\sum_{x_c} \alpha_r \phi_r(x_r) \prod_{(r',s') \in M(r)} m_{r's'}(x_{s'}) = \alpha_s \phi_s(x_s) \prod_{(r'',s'') \in M(s)} m_{r''s''}(x_{s''}) \tag{9.64}$$

where we have replaced the α normalization operator with explicit normalization constants α_r, α_s. Let us now consider the set of messages into s, $M(s)$. Each message (r'', s'') can fall into one of three disjoint categories:

1. $(r'', s'') = (r, s)$
2. $(r'', s'') \in M(s) \cap M(r)$
3. $(r'', s'') \in M(s), (r'', s'') \notin M(r), (r'', s'') \neq (r, s)$.

Note that categories 1 and 2 are disjoint since (r, s) is a message originating in r and thus cannot be in $M(r)$. Let us look at category 3 further. Since $(r'', s'') \in M(s)$, we know that $s'' \subseteq s$ and $(r''\backslash s'') \cap s = \{\}$. We know that s'' is a direct subregion of r'', so $r''\backslash s''$ is a single node, which we will call node d. Hence $\{d\} \cap s = \{\}$, so we know $d \notin s$. Now we know that $(r'', s'') \notin M(r)$, so $(r''\backslash s'') \cap r \neq \{\}$, so $d \in r$. But $d \notin s$, so d must be in $r\backslash s = \{c\}$, so d is the same node as c. Hence $r'' = \{c\} \cup s'' \subseteq \{c\} \cup s = r$, so $r'' \subseteq r$. But we know that our message is not the (r, s) one, so we have strict inclusion, $r'' \subset r$. Hence our message (r'', s'') is a message in

$M(s)$ that originates in a subregion of r, so category 3 is just our previously defined $M(r,s)$.

Using this decomposition of $M(s)$ into three subsets, we can now rewrite (9.64) as follows:

$$\sum_{x_c} \alpha_r \phi_r(x_r) \prod_{(r',s') \in M(r)} m_{r's'}(x_{s'}) = \alpha_s \phi_s(x_s) m_{rs}(x_s) \times \prod_{(r'',s'') \in M(s) \cap M(r)} m_{r''s''}(x_{s''}) \times \prod_{(r'',s'') \in M(r,s)} m_{r''s''}(x_{s''}) \quad . (9.65)$$

All the terms for messages in $M(s) \cap M(r)$ appear on both the left and right hand sides and can be canceled, giving

$$\sum_{x_c} \alpha_r \phi_r(x_r) \prod_{(r',s') \in M(r) \setminus M(s)} m_{r's'}(x_{s'}) = \alpha_s \phi_s(x_s) m_{rs}(x_s) \prod_{(r'',s'') \in M(r,s)} m_{r''s''}(x_{s''}) \quad . (9.66)$$

and hence

$$\sum_{x_c} \frac{\alpha_r}{\alpha_s} \frac{\phi_r(x_r)}{\phi_s(x_s)} \prod_{(r',s') \in M(r) \setminus M(s)} m_{r's'}(x_{s'}) = m_{rs}(x_s) \prod_{(r'',s'') \in M(r,s)} m_{r''s''}(x_{s''}) \quad . (9.67)$$

which leads to

$$\frac{\alpha_r}{\alpha_s} \frac{\sum_{x_c} \frac{\phi_r(x_r)}{\phi_s(x_s)} \prod_{(r',s') \in M(r) \setminus M(s)} m_{r's'}(x_{s'})}{\prod_{(r'',s'') \in M(r,s)} m_{r''s''}(x_{s''})} = m_{rs}(x_s) \quad (9.68)$$

which is essentially (9.60).

9.5. Simulations

Here we present some simulations comparing the performance of decoding some very simple parity-check codes with both the standard LDPC BP-based decoding algorithm and with the generalized BP algorithm. Our first code is about as simple a code as one can get, a rate 2/3 code with two data bits and one parity bit. The parity-check matrix is

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad . \tag{9.69}$$

(Strictly speaking this is not a **low-density** parity-check code, but the standard LDPC decoding algorithm will function even for such codes.) The corresponding network for generalized belief propagation is shown in Fig. 9.2. Nodes 1, 2, and 3 are variable nodes and 0 is the parity-check node. For the generalized BP decoding, we use the set of fundamental Kikuchi regions $R_0 = \{012, 013, 023\}$. The resulting full set of belief propagation update rules are given in Appendix A. We simulated the performance of both algorithms in decoding codewords sent over an additive white Gaussian noise (AWGN) channel at various signal-to-noise ratios (SNR). At each SNR value, we simulated the performance of the decoder over 10^6 codewords. The maximum number of iterations of the algorithm per codeword was set to 30 for both the standard LDPC BP and generalized BP algorithms. The resulting bit error rate (BER) curves are in Fig. 9.3, and the average number of iterations per codeword are in Fig. 9.4. Note that the resulting BER versus SNR curves are identical for both the standard LDPC decoder and the generalized BP decoder. In retrospect, this is not a surprising result, as our network (Fig. 9.2) is a tree, and it is known that the standard belief propagation algorithm will converge to the correct *a posteriori* probabilities in this case. The number of iterations required per codeword is roughly the same for both the standard decoder and the generalized BP decoder, though the iteration count for the latter is slightly higher, which seems a little surprising.

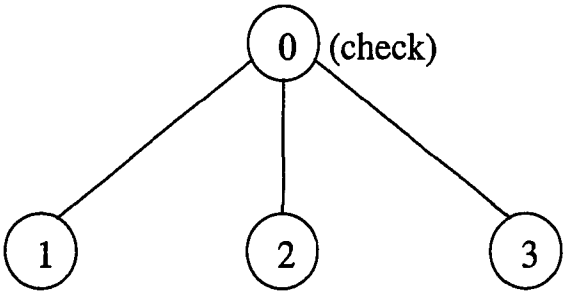


Figure 9.2: Network for our rate 2/3 parity-check code.

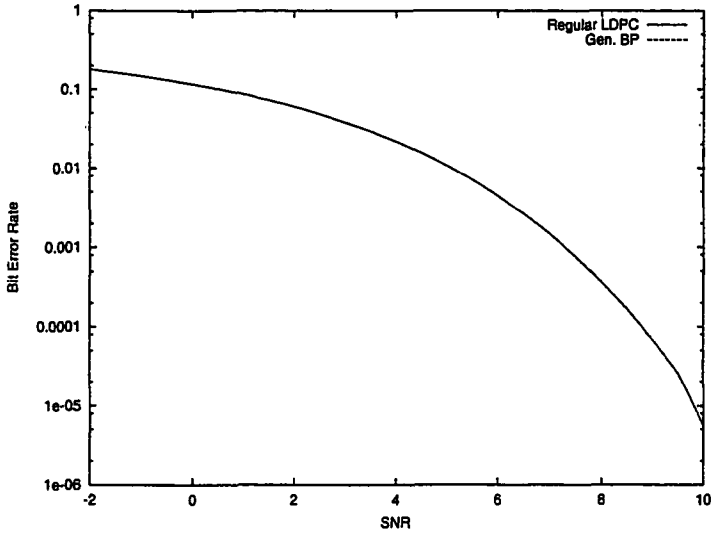


Figure 9.3: Bit error rate of ordinary BP versus generalized BP decoding of rate 2/3 code.

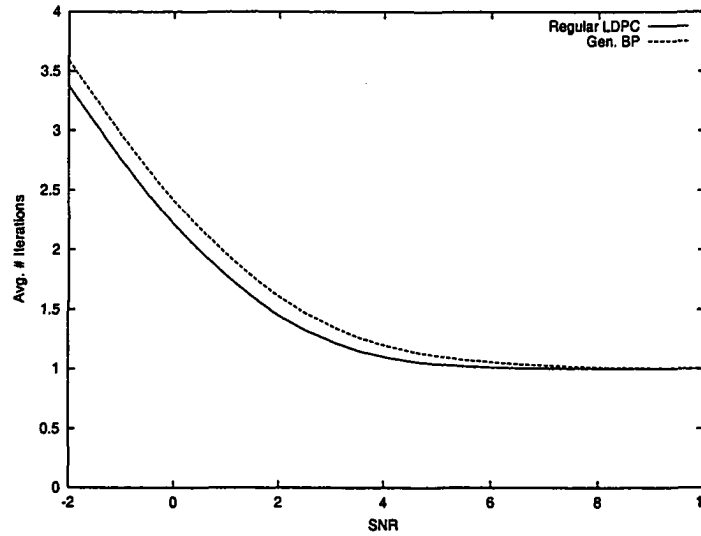


Figure 9.4: Number of iterations needed per codeword for ordinary BP versus generalized BP decoding of rate 2/3 code.

Our next code is a rate 1/2 code with four codeword bits. The parity-check matrix is

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{9.70}$$

and the corresponding network is shown in Fig. 9.5. Here the check nodes are nodes 0 and 5, and our fundamental regions are $R_0 = \{012, 013, 023, 025, 035, 235, 245, 345\}$. Note that the network contains a four-cycle, a closed loop of four nodes. This is different from the previous case where we had a tree-like network. The resulting full set of belief propagation update rules are given in Appendix B. Again, we did simulations of decoding this code with both the regular LDPC algorithm and generalized BP, and the resulting bit error rate curves are in Fig. 9.6, and the average number of iterations per codeword are in Fig. 9.7. Here we see that the generalized BP algorithm does require fewer iterations than the traditional algorithm, with the low SNR cases requiring roughly 1.3 iterations/codeword instead of 4.7. However, the BER performance is worse for generalized BP, with about a 0.7dB loss relative

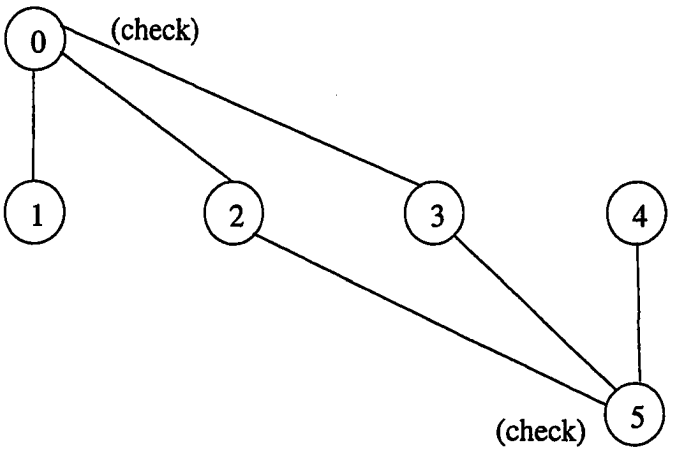


Figure 9.5: Network for our rate 1/2 parity-check code.

to regular LDPC decoding! This is not an encouraging result for those who might want to apply generalized BP to their LDPC codes. Admittedly, this is an extremely small code we are using, and extrapolating performance from it to a more realistic sized code is not something one can be too sure about, but this result does give one cause for concern. (It occurs to us here that, although we know that the Kikuchi free energy is a different estimate of the true free energy, we do not know that it is in fact a more **accurate** estimate than the Bethe free energy. We, and the authors of [29], have been implicitly assuming that it is a better estimate, but we do not know this for a fact. It would be interesting to try and compute the true, Bethe, and Kikuchi free energies on our network and see which is the better estimate; perhaps this should be a topic of further investigation.)

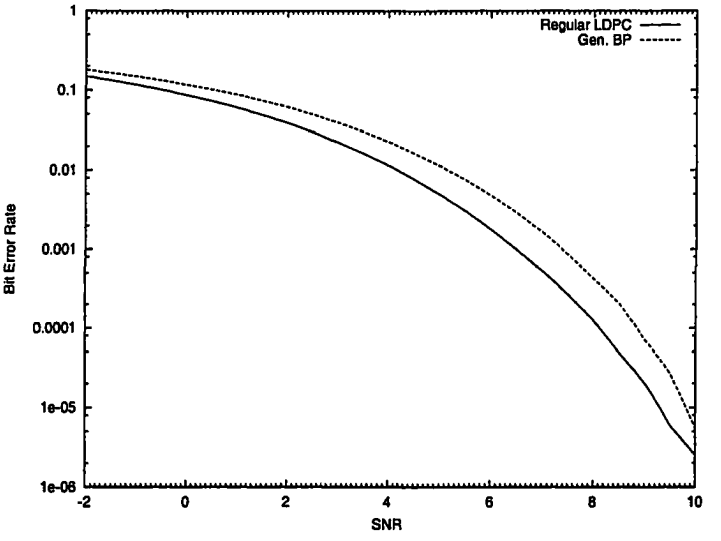


Figure 9.6: Bit error rate of ordinary BP versus generalized BP decoding of rate 1/2 code.

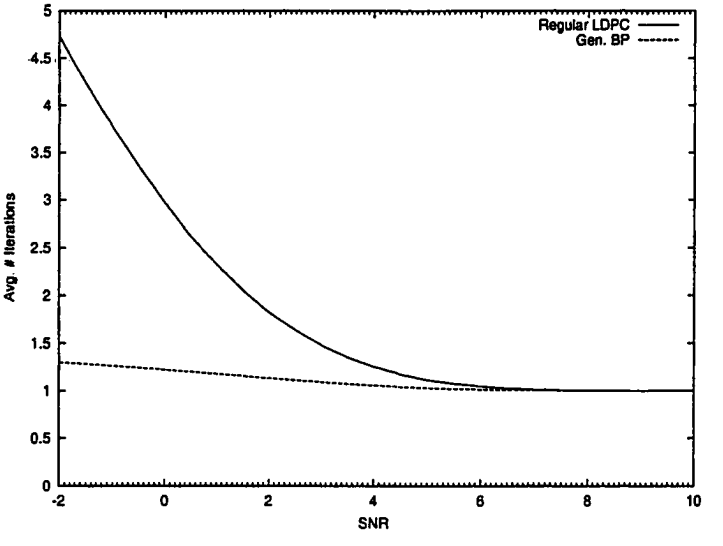


Figure 9.7: Number of iterations needed per codeword for ordinary BP versus generalized BP decoding of rate 1/2 code.

9.6. Applying Generalized Belief Propagation to Real-World LDPC Decoding: Practical Concerns

The reader, having seen the results of generalized belief propagation decoding for simple, almost trivial, codes, is probably wondering how the generalized BP decoder will work on codes with lengths found in the real world. Unfortunately, our opinion is that, barring some major reworking of the algorithm, the use of it with large LDPC codes is not at all practical. We see two main reasons for this, a lesser reason which renders the decoding more problematic (though not impossible), and a second reason which appears to us to be, at present, a major show-stopper.

First, there is the issue of the number of messages we have to keep track of. Consider our simple example rate 2/3 and 1/2 LDPC codes. For the rate 2/3 code with the standard LDPC decoder, we have three binary messages going from the variable nodes to the check nodes and three binary messages going back the other way, for a total of six messages. For the rate 1/2 code, we have a total of 12 messages. But in the generalized BP decoder, the rate 2/3 code requires 12 messages (twice as many), and the rate 1/2 code requires 28 messages (2.33 times as many). The combinatorics of the situation seem to suggest that as we go to bigger and bigger codes the number of messages will increase rapidly with the code size. Let us see if we can estimate what the total number of messages would be for generalized BP decoding of a realistic code.

Consider an LDPC code of codeword length N, with each variable node connected to c_i checks, and suppose the number of checks is L, with each check connected to r_α variable nodes, and suppose that we are, as in the previous section, only doing fundamental regions of size three. Each variable node and check node is going to be the central node in a number of fundamental regions in R_0. The num-

ber of such regions containing variable node i as their central node is the number of combinations of check node peers of i taken two at a time, or $\binom{c_i}{2}$, so the total number of regions in R_0 is

$$\begin{aligned} |R_0| &= \sum_{i=1}^{N} \binom{c_i}{2} + \sum_{\alpha=1}^{L} \binom{r_\alpha}{2} \\ &= \sum_{i=1}^{N} \frac{c_i(c_i-1)}{2} + \sum_{\alpha=1}^{L} \frac{r_\alpha(r_\alpha-1)}{2} \end{aligned} \quad . \tag{9.71}$$

Each of these regions can have two direct connected subregions, so that gives us a number of messages generated from R_0 of

$$|\{(r,s)|r \in R_0\}| = \sum_{i=1}^{N} c_i(c_i-1) + \sum_{\alpha=1}^{L} r_\alpha(r_\alpha-1) \quad . \tag{9.72}$$

Now, the two-node regions in R_1 formed by intersections of pairs of regions in R_1 will just be the set of all the two-node links in the network. Hence

$$|R_1| = \sum_{i=1}^{N} c_i \tag{9.73}$$

and each of these regions gives rise to two subregions and hence two messages, so the total number of messages M_G is

$$M_G = 2\sum_{i=1}^{N} c_i + \sum_{i=1}^{N} c_i(c_i-1) + \sum_{\alpha=1}^{L} r_\alpha(r_\alpha-1) \quad . \tag{9.74}$$

Note that the number of messages required for the standard LDPC decoder is just

$$M_{\text{std}} = 2|R_1| = 2\sum_{i=1}^{N} c_i \quad . \tag{9.75}$$

For a typical LDPC code, such as the one used in [9] for which $N = 4629$ and $L = 277$, we have $M_{\text{std}} = 27826$ and $M_G = 740726$, so the generalized BP algorithm

requires processing about 26.6 times as many messages, and thus at least 26.6 times as much memory and CPU usage per iteration.

Now this by itself would not necessarily render generalized BP impractical, though it does mean that generalized BP would have to render considerable gains in BER to justify the extra resources. However, there is another problem. As we mentioned previously, in this formulation of LDPC decoding as a problem in belief propagation over a network, the check nodes do not have binary states, but instead have states that are bit vectors of length r_α. Hence the number of states each check node can be in is 2^{r_α}. In the simple examples we looked at, r_α was 3, giving nodes with 8 states, but in the code in [9] r_α is always between 50 and 53. Hence every message to a region that includes a check node (and that includes **all** the thousands regions in R_0) must be an array with one dimension being of length at least 2^{50}. This is obviously not practical. The conventional LDPC decoding algorithm avoids this issue by not explicitly considering the check nodes as having such a complex state; in effect, the algorithm does not track the complete state of the bits going into a check node, but only considers whether the bits form a word of even or odd parity. If the generalized BP algorithm is to become practical, one needs a reformulation of it similar to the way the conventional LDPC algorithm is formulated, with messages going back and forth but only passing information about the states of codeword bits, instead of each message giving information about the state of its target nodes. How one is to do this, to successfully "hide" the check node states, when we have messages going towards regions which contain both check nodes and variable nodes, is not at all obvious.

9.7. Conclusion

Yedidia et al. [29] showed that when the BP algorithm converges, it converges to an extremum of the Bethe free energy, and hence to an approximate minimum of the distance (Kullback-Liebler divergence) between the beliefs it computes and the true *a posteriori* probabilities. This is an interesting theoretical result in that it provides some theoretical backing for the validity of the use of BP on networks with cycles. Before this, theory only told us that BP converged to true *a posteriori* results for the limited case of tree-like networks. Yedidia et al. go on to develop a generalization of the BP algorithm based on a generalization of the Bethe free energy defined on multi-node regions of the network. This result is theoretically quite interesting, but not practical to apply to LDPC codes of any substantial size without substantial reworking of the algorithm. Also, the limited testing we have done of the algorithm does not show any improvement in BER performance over the standard LDPC decoder, so it is not known that the generalized BP algorithm would be useful even if it was practical to implement for large LDPC codes.

Chapter 10

The MTR Enforcement Algorithm

10.1. Introduction

As we mentioned back in Chapter 1, practical magnetic recording systems employ some sort of run-length-limited (RLL) or maximum-transition-run (MTR) code to impose restrictions on the kinds of sequences of transitions that are present in the data written on the magnetic medium. These restrictions are imposed in order to make the synchronization circuitry work better and to avoid certain error-prone sequences. Now, any such constrained code, like any code with code rate less than one, imposes a certain amount of redundancy in its output. However, the decoders in the magnetic recording systems we have discussed so far (see, e.g., Fig. 1.10), the BCJR and LDPC decoders, do not take advantage of the redundancy introduced by the MTR constraint. It would be nice if we could modify the decoders to take advantage of this extra redundancy present in our channel inputs. This chapter explains how we can add an extra decoder module, employing what we call the **MTR enforcement algorithm**, and gain a little bit extra performance.

10.2. Where the MTR Enforcer Fits Into the System

Fig. 10.1 shows a diagram of a magnetic recording system with our MTR enforcer added into the system. The enforcer, like the BCJR and LDPC decoder, is a soft-input/soft-output decoder, i.e., its inputs and outputs are log-likelihood values. The flow of the log-likelihood values proceeds as follows:

1. The BCJR decoder creates an initial set of log-likelihood values.

2. We compute the extrinsic information from the BCJR decoder as in Section 3.4. (Note that on the first iteration this is just the BCJR output, as the BCJR log-likelihood inputs are all zeros.)

3. The LDPC decoder operates on this log-likelihood vector and creates new log-likelihood values.

4. The MTR enforcer iterates one or more times on the LDPC decoder's output.

5. The original input to the LDPC decoder is subtracted off from the MTR enforcer output to get the extrinsic information as in Section 3.4. Note that we do the extrinsic information subtraction the same way as in Section 3.4, except that we are effectively treating the LDPC decoder and MTR enforcer as if they were one big decoder.

6. The BCJR and LDPC/MTR extrinsic informations are added to get a total log-likelihood $L(i)$ as in (3.72). The resulting $L(i)$ are hard-decoded and the result checked to see if it is a valid LDPC codeword. If it is, we are done.

7. Otherwise, the extrinsic information is fed back into the BCJR decoder and the process begins again until the limit of number of iterations of turbo equalization is reached.

As in Section 3.4, there is an analogous version of this algorithm without the extrinsic information subtraction; that version goes as follows:

1. The BCJR decoder creates an initial set of log-likelihood values.

2. The LDPC decoder operates on this log-likelihood vector (the BCJR output) and creates new log-likelihood values.

3. The MTR enforcer iterates one or more times on the LDPC decoder's output.

4. The resulting $L(i)$ are hard-decoded and the result checked to see if it is a valid LDPC codeword. If it is, we are done.

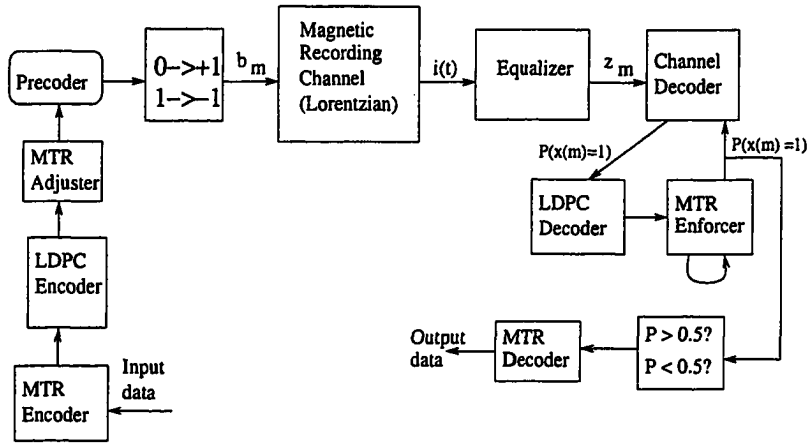


Figure 10.1: Diagram of system for PRML magnetic recording with precoder, MTR code, LDPC code and MTR enforcer.

5. Otherwise, these $L(i)$ values are fed back into the input of the BCJR decoder and the process begins again until the limit of number of iterations of turbo equalization is reached.

In Section 3.4, we saw that whether or not the extrinsic subtraction is done did not make very much difference to the resulting bit error rates of the system. As we shall see later, there is a more substantial difference when the MTR enforcer is added.

The sequence of decoders presented above is admittedly somewhat arbitrary. We argue that intuitively the LDPC decoder should come before the MTR enforcer because the LDPC code is a more powerful code and thus more likely to be able to correct errors. Given the relative looseness of the MTR constraint compared to the constraints of the LDPC parity-check matrix, it seems to us that the MTR enforcer is more likely to be confused than the LDPC decoder (more likely to decode to the wrong MTR codeword) and thus should only see the data after the LDPC decoder has had a chance to clean up as many errors as possible.

10.3. Details of the MTR Enforcer Algorithm

The details of the MTR enforcer algorithm are highly dependent on the particular MTR code in use. We present here the MTR enforcer for the MTR code given in [2]. The MTR enforcer is basically a Bayesian algorithm that computes *a posteriori* log-likelihood ratios given the MTR constraints that the codeword must obey. In the case of the code from [2], this constraint means that we can never have a sequence of four or more one bits in a row, nor can we have twelve or more zero bits. Let us call these two constraints the **One Constraint** and the **Zero Constraint**, respectively. An iteration of the MTR enforcer takes in log-likelihood values $L_{\text{in}}(i)$ and produces outputs $L_{\text{out}}(i)$ as follows.

First, the $L_{\text{in}}(i)$ are converted to probabilities $q_0(i), q_1(i)$ in the obvious way:

$$\begin{aligned} q_0(i) &= \frac{\exp(L_{\text{in}}(i)/2)}{\exp(L_{\text{in}}(i)/2) + \exp(-L_{\text{in}}(i)/2)} \\ q_1(i) &= \frac{\exp(-L_{\text{in}}(i)/2)}{\exp(L_{\text{in}}(i)/2) + \exp(-L_{\text{in}}(i)/2)} \end{aligned} \quad . \tag{10.1}$$

Next, we compute values $NA_{i,j}$ defined as follows: $NA_{i,j}$ is the probability that the four-bit block of bits starting at bit i satisfies the One Constraint (i.e. is not the bit sequence 1111) given that bit $i+j$ is one. In more mathematical language, if we let $\mathbf{x}_{i \to i+3}$ represent the sub-vector of bits from bit i to bit $i+3$, then

$$NA_{i,j} = P(\mathbf{x}_{i \to i+3} \neq [1111] | x_{i+j} = 1) \tag{10.2}$$

is defined for all $j \in \{0, 1, 2, 3\}$ and all i for which the above equation makes sense, that is, $i \in \{0, \ldots, J-4\}$ where J is the length of the input likelihood vector. Note that J may be longer than the original LDPC codeword length N, due to the extra bits stuffed in-between parity bits by the MTR adjuster (Section 1.5.3) to ensure that the LDPC codeword does not violate the MTR constraints. Also

note that we do not have, or need, any variables for the corresponding probabilities $P(\mathbf{x}_{i\to i+3} \neq [1111]|x_{i+j} = 0)$, since by the very nature of the One Constraint, if $x_{i+j} = 0$, the constraint is automatically satisfied. We compute the $NA_{i,j}$ values thus:

$$NA_{i,j} = 1 - \prod_{k\in\{0,1,2,3\}\setminus j} q_1(i+k) \quad . \tag{10.3}$$

Similarly, we define the quantities $NE_{i,j}$ as the probability that the block of bits starting at bit i satisfies the Zero Constraint given that bit $i+j$ is zero, i.e.,

$$NE_{i,j} = P(\mathbf{x}_{i\to i+11} \neq [000000000000]|x_{i+j} = 0) \tag{10.4}$$

for $j \in \{0,\ldots,11\}$ and $i \in \{0,\ldots,J-12\}$. The $NE_{i,j}$ are computed in the obvious fashion:

$$NE_{i,j} = 1 - \prod_{k\in\{0,1,\ldots,11\}\setminus j} q_0(i+k) \quad . \tag{10.5}$$

Then a straightforward Bayesian computation of the *a posteriori* probabilities $\hat{q}_0(i), \hat{q}_1(i)$ of bit i being zero or one respectively given that the Zero Constraint and One Constraint are true over the entire codeword gives us

$$\begin{aligned} \hat{q}_0(i) &= \alpha q_0(i) \prod_{j=0}^{11} NE_{i-j,j} \\ \hat{q}_1(i) &= \alpha q_1(i) \prod_{j=0}^{3} NA_{i-j,j} \end{aligned} \tag{10.6}$$

where α is the usual normalizer to give us total probability of one and, for notational convenience, we have assumed that $NE_{i,j} = 1$ and $NA_{i,j} = 1$ any place where they are not defined above in (10.2) and (10.4). This avoids having to explicitly complicate our equations to handle the cases "at the edges", i.e, when $i - j < 0$. Once $\hat{q}_0(i)$ and $\hat{q}_1(i)$ are computed, we compute the output likelihood $L_{\text{out}}(i)$ in the

obvious fashion:

$$L_{\text{out}}(i) = \log \frac{\hat{q}_0(i)}{\hat{q}_1(i)} \quad . \tag{10.7}$$

10.4. Simulation results

Here we present simulation results of a system such as in Fig. 10.1 with a Lorentzian channel and varying numbers of iterations of the MTR enforcement algorithm. Let us describe here the parameters for the simulation. The magnetic recording channel is Lorentzian with channel density $S = 3.3$. We used the regular rate 0.90 LDPC code from [9], along with the rate 16/17 MTR code from [2]; the Lorentzian channel is equalized to the MEEPR4 $(5+4D-3D^2-4D^3-2D^4)$ response and the $1/(1\oplus D)$ precoder is used, as specified in [2]. Note that the total number of bits input into the channel for each sector is 4996 once the expansion due to the MTR code, the LDPC parity bits, and the extra added bits to keep the overall MTR constraint satisfied are taken into account (1.51). Thus the code rate of the entire system is $4096/4996 = 0.81986$. We used 5 iterations of turbo equalization and an iteration limit of 30 on the LDPC decoder, with a variable number B of iterations of the MTR enforcement algorithm. The signal-to-noise ratio is defined as $SNR = 1/\sigma^2$ where σ^2 is the variance of the noise at the Lorentzian channel output (i.e., **not** the noise at the output of the MEEPR4 equalizer).

Fig. 10.2 shows performance of the system with the number of iterations of MTR enforcement being $B = 0$, 1, 5, and 10 respectively. We use here the variant with the extrinsic information subtraction. Also appearing in the plot is a channel capacity curve. This is not the true capacity bound for the MEEPR4-equalized Lorentzian channel, since we do not know how to calculate that. Instead, it is just the capacity for a perfect MEEPR4 channel computed as in Chapter 7, but with the SNR axis rescaled to bring it in line with the different definition of SNR used

above. It is expected that the true MEEPR4-equalized Lorentzian channel capacity is somewhere close to the ideal MEEPR4 curve, but as this is not known for sure, the curve should be treated as an inexact measure of where the true capacity bound lies. Anyway, the results in Fig. 10.2 show that, unfortunately, the MTR enforcement actually makes the performance worse.

However, Fig. 10.3 tells a much more interesting story. This plot is of the same system as Fig. 10.2, except that this time we use the variant of the system which does **not** do the extrinsic information subtraction. In this case, the system with one round of MTR enforcement ($B = 1$) performs substantially better than the system without the MTR enforcement, providing a gain of about 0.5dB. Increasing B beyond $B = 1$ produces worse performance, so it seems that having only one iteration of MTR enforcement is optimal. This is a very interesting result, since as Fig. 3.2 showed, the presence or absence of the extrinsic subtraction did not make much difference in the system considered there, but not doing the extrinsic subtraction here made a surprising difference with the MTR enforcer, a difference between a -0.25dB loss and a 0.5dB gain. This only reinforces our comments at the end of Chapter 3 that the whole issue of extrinsic information is not understood as well as it should be.

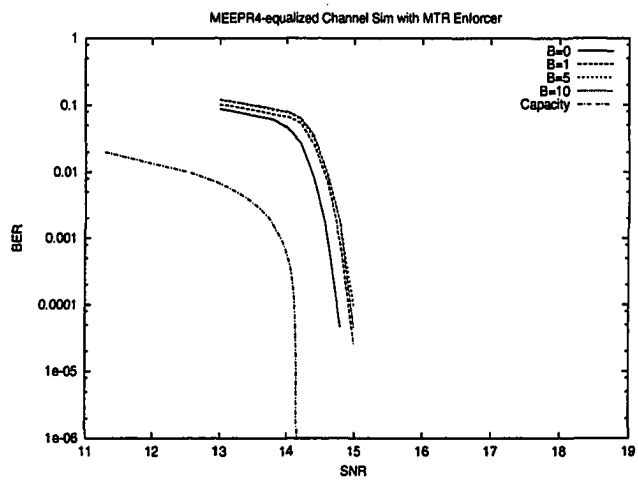


Figure 10.2: Performance of system with varying levels of MTR enforcement, using extrinsic subtraction.

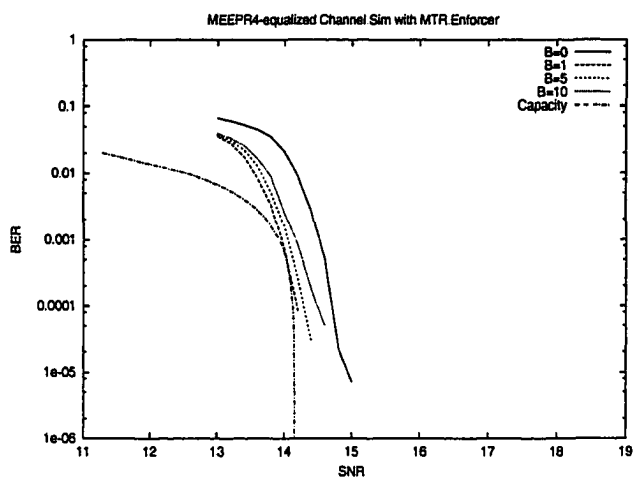


Figure 10.3: Performance of system with varying levels of MTR enforcement, without extrinsic subtraction.

Chapter 11

Conclusions and Further Work

In this dissertation, we have presented a novel method of analyzing the performance of the BCJR algorithm through density evolution, and used this in combination with the known density evolution techniques for LDPC codes to design codes for various partial-response channels associated with magnetic recording. We exhibited codes with performance 0.3 dB to 0.5 dB above that of regular column weight three codes. We also presented a technique, called MTR enforcement, which exploits the knowledge of the maximum-transition-run constraint that MTR-coded data must satisfy to gain a fraction of a dB performance boost.

There are still several open questions to be explored:

1. The proper choice of the d_{lmax}, d_{lmin}, etc., parameters used in the code search is still somewhat unclear. The results we presented in Chapter 8 showed that, generally speaking, lower d_{lmax} means less likelihood of four-cycles appearing in the code and compromising the code performance, but further information on how to choose the d_{lmax} etc. parameters well would be helpful.

2. Density evolution only predicts performance of LDPC codes in the limit of infinite block length and in the absence of cycles (size four or otherwise). Unfortunately, as we have seen, codes that density evolution predicts will do well sometimes do not do so, due to these effects of block length and cycles. It would be nice if someone could develop an extension to density evolution that would take these factors into account.

3. The whole issue of extrinsic information, i.e., should the extrinsic subtraction be performed or not, and why, deserves more attention. As far as we know, this topic has not been discussed much in the literature; most authors seem to assume the extrinsic subtraction should be performed always, and as we saw in Chapters 3 and 10, this may not be the case.

4. One feature of magnetic recording channels that we did not discuss is the issue

of so-called thermal asperities and media defects. Without going into detail here about what they are, the effect of thermal asperities and media defects on the magnetic recording channel is to cause occasional burst erasures of the bit stream, effectively turning the channel into a combination of our existing magnetic recording channel model and an erasure channel. Extending the existing BCJR density evolution to include an erasure effect should be fairly trivial assuming the erasure probability is known, and this should allow one to design codes for magnetic recording systems where erasures may be present. How well the codes would work in practice, given that the erasures occur in bursts and not independently, and that the actual erasure probability may not match that for which the code was designed, remains to be seen.

Bibliography

[1] D. Krueger, "Generalized partial response (GPR) system design for digital magnetic recording," Tech. Rep., CSPLab, The Univ. of Oklahoma, 1995.

[2] T. Nishiya, K. Tsukano, T. Hirai, S. Mita, and T. Nara, "Rate 16/17 maximum transition run (3;11) code on an EEPRML channel with an error-correcting postprocessor," *IEEE Trans. Magn.*, vol. 35, pp. 4378–4385, Sept. 1999.

[3] G. D. Forney, "The Viterbi algorithm," *Proc. IEEE*, vol. 71, pp. 268–278, Mar. 1973.

[4] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

[5] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[6] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, Mar. 1999.

[7] W. E. Ryan, L. L. McPheters, and S. W. McLaughlin, "Combined turbo coding and turbo equalization for PR4-equalized Lorentzian channels," in *Proc. Conf. Inform. Sciences and Systems*, (Princeton, NJ), pp. 489–494, 1998.

[8] B. M. Kurkoski, P. H. Siegel, and J. K. Wolf, "Joint message-passing decoding of LDPC codes and partial-response channels," *IEEE Trans. Inform. Theory*, vol. 48, pp. 1410–1422, June 2002.

[9] H. Song, R. M. Todd, and J. R. Cruz, "Applications of low-density parity-check codes to magnetic recording channels," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 918–923, May 2001.

[10] P. Hodges and D. Cheng, "Large block size for disk drives," tech. rep., National Storage Industry Consortium, 2000. Available at `http://www.nsic.org/large_block_white_paper.pdf`.

[11] R. Johanesson and K. S. Zigangirov, *Fundamentals of Convolutional Coding*. New York, NY: IEEE Press, 1999.

[12] Z. Wu, *Coding and Iterative Detection for Magnetic Recording Channels.* Norwell, Mass.: Kluwer, 2000.

[13] J. L. Fan, A. Friedmann, E. Kurtas, and S. W. McLaughlin, "Low density parity check codes for magnetic recording," *submitted to IEEE J. Select. Areas Commun.*, 2000.

[14] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. Commun.*, (Geneva), pp. 1064–1070, May 1993.

[15] S.-Y. Chung, *On the Construction of Some Capacity-Approaching Coding Schemes.* Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sep. 2000. Available at `http://truth.mit.edu/~sychung/thesis/main.ps.gz`.

[16] J. L. Fan, *Constrained Coding and Soft Iterative Decoding.* Norwell, Mass.: Kluwer Academic Publishers, 2001.

[17] T. J. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, pp. 599–618, Feb. 2001.

[18] P. Hoeher, U. Sorger, and I. Land, "Log-likelihood values and Monte Carlo simulation," in *Proc. Int. Symp. on Turbo Codes and Related Topics*, (Brest, France), pp. 43–46, Sept. 2000.

[19] S.-Y. Chung, T. J. Richardson, and R. Urbanke, "Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation," *IEEE Trans. Inform. Theory*, vol. 47, pp. 657–670, Feb. 2001.

[20] M. Varanasi and B. Aazhang, "Parametric generalized Gaussian density estimation," *J. Acoust. Soc. Amer.*, vol. 86, pp. 1404–1415, October 1989.

[21] F. Yang, R. Tafazolli, B. Evans, L. Luo, and M. Ye, "Performance of iterative turbo decoding with the hypothesis of generalized gaussian distribution for extrinsic values in AWGN and fading channels," in *Proc. IEEE Global Telecommunications Conference (GLOBECOM '01)*, vol. 2, (San Antonio, Texas), pp. 957–962, Nov. 2001.

[22] A. Tesei, R. Bozzano, and C. Regazzoni, "Comparison between asymmetric generalized gaussian (AGG) and symmetric-α-stable (SαS) noise models for signal estimation in non-gaussian environments," in *Proc. IEEE Signal Processign Workshop*, (Banff, Alberta, Canada), pp. 259–263, July 1997.

[23] M. Jeruchim, P. Balaban, and K. Shanmugan, *Simulation of Communication Systems.* New York: Plenum Press, 1992.

[24] D. Arnold and H.-A. Loeliger, "On the information rate of binary-input channels with memory," in *Proceedings IEEE International Conference on Communications*, (Helsinki, Finland), June 2001.

[25] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* New York, NY: John Wiley and Sons, 1991.

[26] J. Hou, P. H. Siegel, and L. B. Milstein, "Performance analysis and code optimization of low density parity-check codes on Rayleigh fading channels," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 924–934, May 2001.

[27] R. J. McEliece, D. J. C. MacKay, and J. F. Chang, "Turbo decoding as an instance of Pearl's belief propagation algorithm," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 140–152, 1998.

[28] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* San Mateo, CA: Morgan Kaufmann, 1988.

[29] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Bethe free energy, Kikuchi approximations, and belief propagation algorithms," Tech. Rep. TR-2001-16, Mitsubishi Electric Research Laboratories, Cambridge, MA, May 2001.

[30] H. A. Bethe, "Statistical theory of superlattices," *Proc. Roy Soc. London A*, vol. 150, pp. 552–575, 1935.

[31] R. Kikuchi, "A theory of cooperative phenomena," *Phys. Rev.*, vol. 81, pp. 988–1003, 1951.

[32] D. J. C. MacKay, J. S. Yedidia, W. T. Freeman, and Y. Weiss, "A conversation about the Bethe free energy and sum-product," Tech. Rep. TR-2001-18, Mitsubishi Electric Research Laboratories, Cambridge, MA, May 2001.

[33] L. Wall and R. L. Schwartz, *Programming Perl.* Sebastapol, CA: O'Reilly and Associates, 1991.

Appendix A

Generalized Belief Propagation Equations for Rate 2/3 LDPC Code

Here we present the full set of generalized belief propagation equations for the network shown in Fig. 9.2. The list of equations was generated from the list of all possible messages with the help of a script in the Perl programming language [33]. A similar script was used to automatically generate the C code for the main iteration subroutine for the simulation program.

$$m_{01\to 0}(x_0) \leftarrow \alpha \sum_{x_1} \frac{\phi_{01}(x_0,x_1)}{\phi_0(x_0)} m_{012\to 01}(x_0,x_1) m_{013\to 01}(x_0,x_1) \tag{A.1}$$

$$\begin{aligned} m_{01\to 1}(x_1) \leftarrow \alpha \sum_{x_0} & \frac{\phi_{01}(x_0,x_1)}{\phi_1(x_1)} m_{02\to 0}(x_0) m_{03\to 0}(x_0) \\ & \times m_{012\to 01}(x_0,x_1) m_{013\to 01}(x_0,x_1) \end{aligned} \tag{A.2}$$

$$m_{02\to 0}(x_0) \leftarrow \alpha \sum_{x_2} \frac{\phi_{02}(x_0,x_2)}{\phi_0(x_0)} m_{012\to 02}(x_0,x_2) m_{023\to 02}(x_0,x_2) \tag{A.3}$$

$$\begin{aligned} m_{02\to 2}(x_2) \leftarrow \alpha \sum_{x_0} & \frac{\phi_{02}(x_0,x_2)}{\phi_2(x_2)} m_{01\to 0}(x_0) m_{03\to 0}(x_0) \\ & \times m_{012\to 02}(x_0,x_2) m_{023\to 02}(x_0,x_2) \end{aligned} \tag{A.4}$$

$$m_{03\to 0}(x_0) \leftarrow \alpha \sum_{x_3} \frac{\phi_{03}(x_0,x_3)}{\phi_0(x_0)} m_{013\to 03}(x_0,x_3) m_{023\to 03}(x_0,x_3) \tag{A.5}$$

$$\begin{aligned} m_{03\to 3}(x_3) \leftarrow \alpha \sum_{x_0} & \frac{\phi_{03}(x_0,x_3)}{\phi_3(x_3)} m_{01\to 0}(x_0) m_{02\to 0}(x_0) \\ & \times m_{013\to 03}(x_0,x_3) m_{023\to 03}(x_0,x_3) \end{aligned} \tag{A.6}$$

$$m_{012\to 01}(x_0,x_1) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{012}(x_0,x_1,x_2)}{\phi_{01}(x_0,x_1)} m_{023\to 02}(x_0,x_2)}{m_{02\to 0}(x_0)} \tag{A.7}$$

$$m_{012\to 02}(x_0,x_2) \leftarrow \alpha \frac{\sum_{x_1} \frac{\phi_{012}(x_0,x_1,x_2)}{\phi_{02}(x_0,x_2)} m_{013\to 01}(x_0,x_1)}{m_{01\to 0}(x_0)} \tag{A.8}$$

$$m_{013\to 01}(x_0,x_1) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{013}(x_0,x_1,x_3)}{\phi_{01}(x_0,x_1)} m_{023\to 03}(x_0,x_3)}{m_{03\to 0}(x_0)} \tag{A.9}$$

$$m_{013\to 03}(x_0,x_3) \leftarrow \alpha \frac{\sum_{x_1} \frac{\phi_{013}(x_0,x_1,x_3)}{\phi_{03}(x_0,x_3)} m_{012\to 01}(x_0,x_1)}{m_{01\to 0}(x_0)} \tag{A.10}$$

$$m_{023\to 02}(x_0,x_2) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{023}(x_0,x_2,x_3)}{\phi_{02}(x_0,x_2)} m_{013\to 03}(x_0,x_3)}{m_{03\to 0}(x_0)} \tag{A.11}$$

$$m_{023\to 03}(x_0,x_3) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{023}(x_0,x_2,x_3)}{\phi_{03}(x_0,x_3)} m_{012\to 02}(x_0,x_2)}{m_{02\to 0}(x_0)} \tag{A.12}$$

$$b_1(x_1) \leftarrow \psi_1(x_1) m_{01\to 1}(x_1) \tag{A.13}$$

$$b_2(x_2) \leftarrow \psi_2(x_2) m_{02\to 2}(x_2) \tag{A.14}$$

$$b_3(x_3) \leftarrow \psi_3(x_3) m_{03\to 3}(x_3) \tag{A.15}$$

Appendix B

Generalized BP Equations for Rate 1/2 LDPC Code

Here we present the full set of generalized belief propagation equations for the network shown in Fig. 9.5. As with the previous code, the list of equations was generated from the list of all possible messages with a Perl script, and a similar Perl script was used to automatically generate the C code for the main iteration subroutine for the simulation program.

$$m_{012\to01}(x_0,x_1) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{012}(x_0,x_1,x_2)}{\phi_{01}(x_0,x_1)} m_{023\to02}(x_0,x_2) m_{025\to02}(x_0,x_2) m_{25\to2}(x_2)}{m_{02\to0}(x_0)} \tag{B.1}$$

$$m_{012\to02}(x_0,x_2) \leftarrow \alpha \frac{\sum_{x_1} \frac{\phi_{012}(x_0,x_1,x_2)}{\phi_{02}(x_0,x_2)} m_{013\to01}(x_0,x_1)}{m_{01\to0}(x_0)} \tag{B.2}$$

$$m_{013\to01}(x_0,x_1) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{013}(x_0,x_1,x_3)}{\phi_{01}(x_0,x_1)} m_{023\to03}(x_0,x_3) m_{035\to03}(x_0,x_3) m_{35\to3}(x_3)}{m_{03\to0}(x_0)} \tag{B.3}$$

$$m_{013\to03}(x_0,x_3) \leftarrow \alpha \frac{\sum_{x_1} \frac{\phi_{013}(x_0,x_1,x_3)}{\phi_{03}(x_0,x_3)} m_{012\to01}(x_0,x_1)}{m_{01\to0}(x_0)} \tag{B.4}$$

$$m_{023\to02}(x_0,x_2) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{023}(x_0,x_2,x_3)}{\phi_{02}(x_0,x_2)} m_{013\to03}(x_0,x_3) m_{035\to03}(x_0,x_3) m_{35\to3}(x_3)}{m_{03\to0}(x_0)} \tag{B.5}$$

$$m_{023\to03}(x_0,x_3) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{023}(x_0,x_2,x_3)}{\phi_{03}(x_0,x_3)} m_{012\to02}(x_0,x_2) m_{025\to02}(x_0,x_2) m_{25\to2}(x_2)}{m_{02\to0}(x_0)} \tag{B.6}$$

$$m_{025\to02}(x_0,x_2) \leftarrow \alpha \frac{\sum_{x_5} \frac{\phi_{025}(x_0,x_2,x_5)}{\phi_{02}(x_0,x_2)} m_{235\to25}(x_2,x_5) m_{245\to25}(x_2,x_5) m_{35\to5}(x_5) m_{45\to5}(x_5)}{m_{25\to2}(x_2)} \tag{B.7}$$

$$m_{025\to25}(x_2,x_5) \leftarrow \alpha \frac{\sum_{x_0} \frac{\phi_{025}(x_0,x_2,x_5)}{\phi_{25}(x_2,x_5)} m_{012\to02}(x_0,x_2) m_{023\to02}(x_0,x_2) m_{01\to0}(x_0) m_{03\to0}(x_0)}{m_{02\to2}(x_2)} \tag{B.8}$$

$$m_{035\to03}(x_0,x_3) \leftarrow \alpha \frac{\sum_{x_5} \frac{\phi_{035}(x_0,x_3,x_5)}{\phi_{03}(x_0,x_3)} m_{235\to35}(x_3,x_5) m_{345\to35}(x_3,x_5) m_{25\to5}(x_5) m_{45\to5}(x_5)}{m_{35\to3}(x_3)} \tag{B.9}$$

$$m_{035\to35}(x_3,x_5) \leftarrow \alpha \frac{\sum_{x_0} \frac{\phi_{035}(x_0,x_3,x_5)}{\phi_{35}(x_3,x_5)} m_{013\to03}(x_0,x_3) m_{023\to03}(x_0,x_3) m_{01\to0}(x_0) m_{02\to0}(x_0)}{m_{03\to3}(x_3)} \tag{B.10}$$

$$m_{235\to25}(x_2,x_5) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{235}(x_2,x_3,x_5)}{\phi_{25}(x_2,x_5)} m_{035\to35}(x_3,x_5) m_{345\to35}(x_3,x_5) m_{03\to3}(x_3)}{m_{35\to5}(x_5)} \tag{B.11}$$

$$m_{235\to35}(x_3,x_5) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{235}(x_2,x_3,x_5)}{\phi_{35}(x_3,x_5)} m_{025\to25}(x_2,x_5) m_{245\to25}(x_2,x_5) m_{02\to2}(x_2)}{m_{25\to5}(x_5)} \tag{B.12}$$

$$m_{245\to25}(x_2,x_5) \leftarrow \alpha \frac{\sum_{x_4} \frac{\phi_{245}(x_2,x_4,x_5)}{\phi_{25}(x_2,x_5)} m_{345\to45}(x_4,x_5)}{m_{45\to5}(x_5)} \tag{B.13}$$

$$m_{245\to45}(x_4,x_5) \leftarrow \alpha \frac{\sum_{x_2} \frac{\phi_{245}(x_2,x_4,x_5)}{\phi_{45}(x_4,x_5)} m_{025\to25}(x_2,x_5) m_{235\to25}(x_2,x_5) m_{02\to2}(x_2)}{m_{25\to5}(x_5)} \tag{B.14}$$

$$m_{345\to35}(x_3,x_5) \leftarrow \alpha \frac{\sum_{x_4} \frac{\phi_{345}(x_3,x_4,x_5)}{\phi_{35}(x_3,x_5)} m_{245\to45}(x_4,x_5)}{m_{45\to5}(x_5)} \tag{B.15}$$

$$m_{345\to45}(x_4,x_5) \leftarrow \alpha \frac{\sum_{x_3} \frac{\phi_{345}(x_3,x_4,x_5)}{\phi_{45}(x_4,x_5)} m_{035\to35}(x_3,x_5) m_{235\to35}(x_3,x_5) m_{03\to3}(x_3)}{m_{35\to5}(x_5)} \tag{B.16}$$

$$m_{01\to0}(x_0) \leftarrow \alpha \sum_{x_1} \frac{\phi_{01}(x_0,x_1)}{\phi_0(x_0)} m_{012\to01}(x_0,x_1) m_{013\to01}(x_0,x_1) \tag{B.17}$$

$$m_{01\to1}(x_1) \leftarrow \alpha \sum_{x_0} \frac{\phi_{01}(x_0,x_1)}{\phi_1(x_1)} m_{012\to01}(x_0,x_1) m_{013\to01}(x_0,x_1) m_{02\to0}(x_0) m_{03\to0}(x_0) \tag{B.18}$$

$$\begin{aligned} m_{02\to0}(x_0) \leftarrow \alpha \sum_{x_2} & \frac{\phi_{02}(x_0,x_2)}{\phi_0(x_0)} m_{012\to02}(x_0,x_2) m_{023\to02}(x_0,x_2) \\ & \times m_{025\to02}(x_0,x_2) m_{25\to2}(x_2) \end{aligned} \tag{B.19}$$

$$\begin{aligned} m_{02\to2}(x_2) \leftarrow \alpha \sum_{x_0} & \frac{\phi_{02}(x_0,x_2)}{\phi_2(x_2)} m_{012\to02}(x_0,x_2) m_{023\to02}(x_0,x_2) m_{025\to02}(x_0,x_2) \\ & \times m_{01\to0}(x_0) m_{03\to0}(x_0) \end{aligned} \tag{B.20}$$

$$m_{03\to 0}(x_0) \leftarrow \alpha \sum_{x_3} \frac{\phi_{03}(x_0,x_3)}{\phi_0(x_0)} m_{013\to 03}(x_0,x_3) m_{023\to 03}(x_0,x_3) \times m_{035\to 03}(x_0,x_3) m_{35\to 3}(x_3) \tag{B.21}$$

$$m_{03\to 3}(x_3) \leftarrow \alpha \sum_{x_0} \frac{\phi_{03}(x_0,x_3)}{\phi_3(x_3)} m_{013\to 03}(x_0,x_3) m_{023\to 03}(x_0,x_3) m_{035\to 03}(x_0,x_3) \times m_{01\to 0}(x_0) m_{02\to 0}(x_0) \tag{B.22}$$

$$m_{25\to 2}(x_2) \leftarrow \alpha \sum_{x_5} \frac{\phi_{25}(x_2,x_5)}{\phi_2(x_2)} m_{025\to 25}(x_2,x_5) m_{235\to 25}(x_2,x_5) m_{245\to 25}(x_2,x_5) \times m_{35\to 5}(x_5) m_{45\to 5}(x_5) \tag{B.23}$$

$$m_{25\to 5}(x_5) \leftarrow \alpha \sum_{x_2} \frac{\phi_{25}(x_2,x_5)}{\phi_5(x_5)} m_{025\to 25}(x_2,x_5) m_{235\to 25}(x_2,x_5) m_{245\to 25}(x_2,x_5) \times m_{02\to 2}(x_2) \tag{B.24}$$

$$m_{35\to 3}(x_3) \leftarrow \alpha \sum_{x_5} \frac{\phi_{35}(x_3,x_5)}{\phi_3(x_3)} m_{035\to 35}(x_3,x_5) m_{235\to 35}(x_3,x_5) \times m_{345\to 35}(x_3,x_5) m_{25\to 5}(x_5) m_{45\to 5}(x_5) \tag{B.25}$$

$$m_{35\to5}(x_5) \leftarrow \alpha \sum_{x_3} \frac{\phi_{35}(x_3,x_5)}{\phi_5(x_5)} m_{035\to35}(x_3,x_5) m_{235\to35}(x_3,x_5) m_{345\to35}(x_3,x_5) \times m_{03\to3}(x_3) \tag{B.26}$$

$$m_{45\to4}(x_4) \leftarrow \alpha \sum_{x_5} \frac{\phi_{45}(x_4,x_5)}{\phi_4(x_4)} m_{245\to45}(x_4,x_5) m_{345\to45}(x_4,x_5) m_{25\to5}(x_5) m_{35\to5}(x_5) \tag{B.27}$$

$$m_{45\to5}(x_5) \leftarrow \alpha \sum_{x_4} \frac{\phi_{45}(x_4,x_5)}{\phi_5(x_5)} m_{245\to45}(x_4,x_5) m_{345\to45}(x_4,x_5) \tag{B.28}$$

$$b_1(x_1) \leftarrow \psi_1(x_1) m_{01\to1}(x_1) \tag{B.29}$$

$$b_2(x_2) \leftarrow \psi_2(x_2) m_{02\to2}(x_2) m_{25\to2}(x_2) \tag{B.30}$$

$$b_3(x_3) \leftarrow \psi_3(x_3) m_{03\to3}(x_3) m_{35\to3}(x_3) \tag{B.31}$$

$$b_4(x_4) \leftarrow \psi_4(x_4) m_{45\to4}(x_4) \tag{B.32}$$